TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

NGUYEN CAO THI THANH

DESIGN AND IMPLEMENTATION OF A JAVASCRIPT SENSOR FRAMEWORK API

Master of Science thesis

Examiner: Prof. Kari Systä
Examiner and topic approved by the Faculty Council of the Faculty of Computer and Electrical Engineering
on January 2017

ABSTRACT

NGUYEN CAO THI THANH: Design and implementation of a JavaScript Sensor Framework API
Tampere University of technology
Master of Science Thesis, 62 pages
Month and year of completion: January 2017
Master's Degree in Information Technology
Major: Pervasive System
Examiner: Professor Kari Systä

Keywords: Internet of Things (IoT), Sensors, Sensor framework, Application Programming Interface (API), JavaScript

Along with the development of Internet of Things, an increasing number of sensors are being exposed on Web environment. However, most of the current sensor libraries are created for particular purposes. Therefore, they are incompatible with each other. This reality creates an inconvenient environment for developers to implement their Web applications.

This thesis proposes a design and implementation of a sensor framework API which were written in JavaScript. This design was done based on analyses of different existing sensor APIs and the Generic Sensor API Specification of World Wide Web Consortium organization. The research was conducted by analyzing their open-source code and specifications available on their GitHub and websites. The objective was to review and compare advantages and drawbacks between different approaches. These information created a foundation for designing a good sensor framework API.

The implementation of the sensor framework has been written in JavaScript, built on the Node.js platform and tested on Raspberry Pi board Model B. Detailed instructions of using and extending the framework are presented in this thesis. Testing has been done with temperature sensors, including Raspberry Pi on-board temperature sensor, and two other DS18B20 sensors as well.

PREFACE

This Master of Science thesis work, "Design and implementation of a Sensor Framework API" has been written and finished as a final part of my M.Sc Degree in Department of Pervasive Computing at Tampere University of Technology (TUT), Tampere, Finland.

Firstly, I would like to express my sincere gratitude to my supervisor and examiner Prof. Kari Systä for his continuous support and guidance in my thesis work at the university. He also gave me lots of patience, motivation, and knowledge which helped me in all the time of research and writing this thesis. Besides, I would like to thanks Prof. Tommy Mikkonen for giving me the opportunity to work on this thesis.

My sincere thanks also goes to Sriharsha Vathsavayi for his valuable review, comments, and discussions about my design and implementation. I also thanks all my friends at TUT for the precious time we have studied and had fun together.

Finally, I take this opportunity to express my deep gratitude to my family for their love, unfailing encouragement and support. This accomplishment would not have been possible without them.

Tampere, November, 2016

Cao Thi Thanh Nguyen

CONTENTS

LIST OF FIGURES

LIST OF PROGRAMS

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
DOM	Document Object Model
FIFO	First In First Out queue
GPS	Global Positioning System
GUI	Graphical User Interface
I2C	Inter-Intergrated Circuit
IDE	Integrated Development Environment
IO	Iput/Ouput
IoT	Internet of Things
npm	Node.js Package Manager
PLM	Power-Line Modem
Rev	Revision
RFID	Radio-frequency Identification
SoC	System on Chip
SPI	Serial Peripheral Interface bus
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
W3C	World Wide Web Consortium
WD	Working Draft

1. INTRODUCTION

The development of communication devices and wireless network technologies is paving the way for the emerging of Internet of Things (IoT). IoT is evaluated as an important trend which has great effects on human life. IoT is making ‘things’ more intelligent by connecting them together and allows them to extract data from environment. These data can be processed automatically by applying algorithms and shared over the Internet. It has potential to create an intelligent environment with unlimited number of devices communicating with each other.

Meanwhile, sensors play an important part in IoT technology. It helps ‘things’ measure different environmental factors, such as: temperature, humidity, pressure, geographical location, etc. With the measured data measurement, physical environment is easier to observe, understand, and forecast. In fact, there is a large number of sensors used in different applications nowadays. They are diverse in functionality, type, and operation. Currently, there is no consistent way to handle different sensor types and expose different sensor data to the Web. Most of sensor APIs available are ad-hoc, incompatible, and unstandardized. This situation creates burden for developers to access to different resources.

Using same API for general connection/disconnection, handling events, checking capacity and compatibility for all sensors provides a friendly and easy way for implementing applications. Developers gain huge benefits if all sensor-based APIs were consistent. Consistent API framework reduces the cost of development, integration, and maintenance. This approach also gives a possibility to apply the same APIs on different sensors available on different boards.

Furthermore, a sensor framework creates a convenient environment for devices to update dynamically and extended for new types of application. In this case, a framework can play a role of standard structure helping the application to fit in the IoT system. It provides instructions to create a template for application. This task may be handled by the IDE. Then, developers need to fill necessary code specifying in this template [17]. By using the same template as a consistent interface to implement necessary functions and events, applications or sensor libraries are easy to be managed, registered, queried, and automatically deployed in IoT system.

Those are reasons of why developing a consistent sensor framework API will promote consistency across sensor APIs and bring huge benefits to Web application developers.

The purpose of my thesis is to design and implement an API framework for sensors which is written in JavaScript. This framework should meet some basic requirements. These requirements are selected based on the evaluation of different API designs. The scope of the thesis is focusing on the API itself. Other issues such as: bootstrapping, troubleshooting, security, etc. are out of scope.

The thesis is organized as follows: Chapter 1 introduces the subject, objective and outline of the thesis. Chapter 2 presents theoretical background about Internet of Things and sensors. It goes through IoT concepts, characteristics, and applications in different areas of life. It explains the role of sensors in the IoT development and provides basic knowledge about sensors such as definition, characteristics and classification. Chapter 3 contains analyses of different approaches to defining Sensor API. It also justifies the benefits of developing a sensor framework API and proposes basic requirements of a good design. Chapter 4 goes into details on my sensor framework API design and implementation. Instructions of using and extending the framework are also explained in this chapter. Chapter 5 contains the assessment and testing phase of the framework. Finally, chapter 6 draws some conclusions about the thesis and proposes the possible future work.

2. THEORETICAL BACKGROUND

This chapter provides a foundation of the key concepts relating to the thesis. It gives in detail explanation of the Internet of Things concept, its characteristics and applications, and its huge impact on human life. The chapter also covers basic knowledge about sensors and their role in the development of IoT technology.

2.1 Internet of Things (IoT)

The term of Internet of Things (IoT), first proposed in 1999, has emerged during last years of 20th century. Kevin Ashton, the British technology pioneer who was the first used the term "Internet of Things" defined it as a system "*in which objects in the physical world could be connected to the Internet by sensors*" [1]. In IoT, with the help of sensors, every object in everyday life can connect, gather, react, and exchange data through a network.

Only within few recent years, the Internet of Things has become a hot trend and gained attention from scientists all over the world because of its huge potential effect on human life. IoT allows "*people and things to be connected Anytime, Anyplace, with Anything and Anyone, ideally using Anypath/network and Any services*" [2]. That means it has potential to connect all 'Things' in life through the Internet and control almost aspects of life.

2.1.1 Characteristics

These are six characteristics of IoT that Carlos Elena-Lenz mentioned in his article in 2014 [3]. These characteristics provide a comprehensive description about IoT's major features.

- **Intelligence**: An IoT system not only gathers information from the outside world, but also uses algorithms to process collected data, in order to give automatic instructions to react to detected environmental changes. Automatic response is a major characteristic of IoT, which gives it a capability called "intelligence". Nowadays, AI (Artificial Intelligence) development gives IoT a big advantage to heighten its intelligence and creates more and more smart systems day by day.
- **Connectivity**: It is an IoT's natural characteristic which was defined in its definition. When IoT reaches its full potential, it can create a network of unlimited number of devices. Devices connected in IoT can be either wired or wireless

ones. Because IoT has a wide range of devices to support, connectivity requirements of IoT are so diverse. It is difficult to have a single technology which satisfies all connectivity requirements such as: power, range, cost, etc. Besides, the diversity of devices in IoT network also presents difficulties in dedvice and data management.

- **Sensing**: With the help of sensors, IoT systems have ability to sense the physical world around them, such as: temperature, distance, humid, sound, etc. Data gathering from sensors provides rich information for people to understand more accurately about the world.
- **Expressing**: IoT creates a new way to interact between users and their real world. By concentrating in data, providing intelligent algorithms, and applying GUI technologies, people can control their real world in a more effective, rapid, and accurate way.
- **Energy**: The scenario for the IoT is to bring 'billion things or more' acting together. Such huge things like this consume tons of energy. Energy requirements in IoT system includes energy harvesting, power efficiency, and charging infrastructure [3].
- **Safety**: With such huge system like IoT, the impact it brings to human life is unpredictable. It means IoT can cause serious problems if it is unsafe. Safety here covers both end points of data transaction and throughout network to secure personal data from attacking, stealing, and being lost. Moreover, physical well-being security needs to be included in safety usability when designing.

2.1.2 Applications

IoT field is wide and full of applications. IoT applications can be found in many aspects of life such as transportation, manufacturing, management, media, medical, healthcare systems, etc. Here, I list some of noticeable applications that demonstrate the enormous influence of IoT on human life, based on list 'Top 50 Internet of Things Applications'posted at the Libelium website [4].

- **Smart cities**: IoT can take part in building smart parking in which citizens can monitor parking space available in the city. At the same time, Traffic Congestion System helps them to track traffic situation around a city and optimize their driving and walking routes. Other aspects such as: noise, free WiFi stations, buses, weather conditions, waste, etc. can be monitored and reported in real-time. You can think about a Smart Lighting System which manages street lights to adapt automatically to weather conditions. In this field, rich of applications can be come up with, making citizens' life easier and more comfortable.
- **Smart Environment**: IoT helps to detect wildfire, even at the early stage by monitoring combustion gases and preemptive fire conditions. Soil moisture, vibration, and earth density can be observed to detect dangerous patterns in land

conditions such as avalanche or landslide. Besides, data about earthquakes and air pollution also can be gathered and processed to detect and help to make appropriate decisions.

- **Smart Agriculture**: Regarding to agriculture field, IoT is currently used in Wine Quality Controlling. This application monitors data relating to soil moisture, trunk diameter of vineyards, amount of sugar, and level of fermentation. These data are collected and calculated to manage and control wine quality. Farmers can build automatic control greenhouses, or, even in large farms. Sensors located throughout the farms will calculate level of necessary water and fertilizer, detect possible diseases and pests, and estimate potential yearly productivity.
- **Healthcare**: Some smart systems have been built to improve patients' health condition. Systems can be installed in houses to assist elderly or disable people to live more independently. At the same time, hospitals are gradually taking advantage of IoT development to build smart systems which control and manage medicine, organic elements, or vaccines, and monitor patients' conditions.

IoT is also seen in other fields, such as Retail, Industrial Control, Logistics, Security & Emergency, Environment Measurement, etc.

IoT is affecting human society at a fast pace. There are some striking statistics which can prove it. According to Juniper Research, from 2015 to 2020, the number of devices connected is projected to increase from 13.4 billion to over 38 billion [5]. Furthermore, BBC Research predicted that the value of global sensor market will reach 154.4 billion dollars by 2020 [6]. The McKinsey Global Institute even gave a more impressive prediction with $11 trillion which is the IoT's total economic impact up to 2025 [7]. These statistics data is enough to bring the IoT to attention and highlight the magnitude of the IoT in the present time as well as in the future. So why have many economists given IoT such profound forecast? The reason can be found in the list of IoT applications in the section 2.1.2, which just covers some parts of the whole picture. Many scientists and economists believe that just a decade from now, people will be heavily dependent on information derived from continuous stream of data from IoT systems. IoT becomes the lodestar for technical innovation and promises to create more jobs. The boom of IoT can be compared to the boom of automobile or electric devices. People soon lose their ability to manage their life without smart devices, applications, and systems. Human society therefore has been changing its appearance; people have been changing the way they live; companies have been changing the way they operate. We will see that scenes in fiction films is no longer far away.

On the other hand, huge prospect usually comes along with serious risks. Same here, IoT presents important questions for its security, privacy, and safety. Reliance on connected devices brings major risks of security and privacy, especially in fields holding users' personal and sensitive information such as health, financial and proprietary data.

Security should be considered first and foremost ahead of product launch. In addition, many IoT applications directly affect the safety of many customers such as traffic or healthcare applications. Therefore, safety is the second line that IoT's standards should pass.

Currently, the significant downside of this trend is that we currently lack of compliance standards when facing with a huge data networks to manage and a large different sets of device standard connected in. Consequences of this deficiency are high risks of data breach and low efficiency.

Summary, IoT creates new enormous opportunities along with new risks to human life. However, this trend is unstoppable and our missions are to predict and prepare all we could for this technical tsunami.

2.2 Sensors

Before the birth of IoT's concept, sensors had been widely used in many devices a long time ago. Sensors can be used in daily-used devices such as home applications as well as special-used equipment in hospital or industry. For example, only in washing machines, we can find temperature sensor, water level sensor, leak detector sensor, etc. Or, a refrigerator is installed different types of sensor, such as: pressure sensor, temperature sensor, humidity sensor, etc.

2.2.1 Definition

Definition of sensor:

"*A sensor is an object whose purpose is to detect events or changes in its environment, and then provide a corresponding output*."[8]

In brief, sensor can be seen as an object translating specific physical environmental parameters into signals that can be measured electrically. If the signal is used in digital equipment, an analog-digital-converter is applied to convert analog signal into digital data. Then, the data is gathered, analyzed, exchanged, or applied algorithms to produce appropriate output that is suitable to specific application or system.

2.2.2 Characteristics

Errors

In an ideal situation, a good sensor is highly sensitive to environmental factor it measures. It means that this sensor is able to recognize slight changes of that property in its environment, thus providing high accurate measurements. In addition, a good sensor

should not be sensitive to other environmental properties it does not measure. If it has, those properties will affect and distort the result. Moreover, the accuracy also requires the integrity of the environment. Therefore, the operation of sensors should not change their ambient environment in which they are working [8].

Unfortunately, an ideal situation is hard to achieve. So, developers need to accept some kinds of deviation generated by sensors. Some common deviations usually can be observed are:

- **Noise**: happens when the deviation has changed over time. It causes the output data to become inaccurate.
- **Sensitive deviation**: is the difference between measured values from the real ones.
- **Range**: is the limitation of the maximum and minimum values that a sensor can measured. Out of this scale, the sensor is unable to work properly anymore.
- **Nonlinearity**: happens when the sensitivity changes over the full range of the sensor.
- **Limited sampling frequency**: happens when the signal created by sensors is measured digitally. Working out of sampling frequency will provide inaccurate data.
- **Digitization error**: When the output of sensor is converted to digital data, these data suffer a kind of error called digitization error caused by the approximation of the measured property.

Resolution

Besides deviation, resolution is another important feature of a sensor. It describes the smallest change that sensor can detect from its environment. The smaller resolution is, the more accurate the data is. In other word, resolution can be seen as the precision of the measurement.

Calibration

As we can see in the above section, there are many factors affecting the precision of a sensor. In fact, there is no perfect sensor. Therefore, the important reason of calibrating a sensor is to achieve the best possible accuracy.

The first thing you need when calibrating a sensor is having a standard reference. A standard reference can be a calibrated sensor or a standard physical reference. Firstly, a calibrated sensor is a sensor that is guaranteed to be accurate. It will be used to make comparison against other sensors. That calibrating information is included in documentation as well as any adjustment that should be added into the output to make the result more reliable. Secondly, a standard physical reference is a physical standard which can be used as a standard reference, such as: rulers, thermometer, meter, etc. In other cases,

some physical phenomena can be used as standard references as well. For example, 100°C is the boiling temperature of water at sea level or gravity on the surface of the earth is always equal to 1G [21].

2.2.3 Classification

There are many ways to classify sensors. However, people typically distinguish them based on their function or property. Some examples is given in the list below [22]:

- **Acoustic and Sound Sensors**
  e.g.: microphone, hydrophone
- **Automotive Sensors**
  e.g.: speedometer, fuel ratio meter
- **Chemical Sensors**
  e.g.: PH sensors, sensors used to detect the presence of specific substance
- **Electric and Magnetic Sensors**
  e.g.: metal detector, galvanometer (to detect and measure small electric current), hall sensor (measures flux density)
- **Environmental Sensors**
  e.g.: rain gauge, moisture and humidity sensor
- **Optical Sensors**
  e.g.: photo diode, photo transistor
- **Mechanical Sensors**
  e.g.: strain gauge, potential meter (measures displacement)
- **Thermal and Temperature Sensors**
  e.g.: calorimeter, thermocouple, and thermistor
- **Proximity and Presences Sensors**
  e.g.: motion detector

Besides, sensors can be classified in other ways, based on:

- **Application**: industrial or non-industrial sensors
- **Output format**: analog or digital sensors
- **Requirement of power supply**: active or passive sensors

Furthermore, W3C (The World Wide Web Consortium) mentioned another way to classify sensors, which are low-level and high-level sensors [9]. To be more specific, low-level sensor refers to sensors that are characterized by its implementation technology, such as gyroscope sensor. On the other hand, high-level sensor is characterized by its function, regardless of the implementation. Implementation of a high-level sensor is usually complicated. Its readings depend on data derived from different low-level sen-

sors. The output of a high-level sensor is the result of the process of applying algorithms on those data. For example, geolocation sensors, which provides location information of users, apply triangulation algorithm onto data gathered from GPS sensors. The process of combining different readings from different sensors is called sensor fusion.

In brief, there are many methods to classify sensors, based on different objectives. Even belonging in the same category, sensors can be classified into smaller groups based on other dimensions. Such as photoelectric sensors can be grouped by structures, sensing modes, beam sources, or output circuits, etc. Classification schemes range from simple to complex, depending on concepts chosen to classify. However, classification only reflects limited aspects of a sensor. Therefore, the best way to study a sensor is examining all its properties such as specification, physical feature, stimulus, output type, and application field, etc.

3. ANALYSIS OF EXISTING SENSOR APIS

In order to develop a framework API, having an overview of different sensor APIs is a crucial task (especially ones developed on top of Node.js platform). Therefore, I've studied different libraries supported for different development boards with their own external sensors such as Phidgets, Tessel, and Insteon Hub. Besides, native platforms such as Android and iOS are also included in the menu by their full-fledged development and rich history in working with sensors. Last but not least, W3C's specification is the most important movement in this field so far. Thus, I've devoted my time not only to go through details of its draft but also discussions and debates between W3C's editors, developers around open issues relating to this project.

Based on these knowledge, I compare these Sensor APIs by picking up different approaches and implementation styles. I also present their advantages and drawbacks from different views, including: suitability, flexibility, and efficiency.

3.1 JavaScript Sensor APIs

I've been researched three JavaScript APIs: Phidgets API, Tessel API, and Insteon Hub API. Each of these APIs are shown in detail respectively in this section, following the same organized way: introduction, API features, and significant points in their implementation.

3.1.1 Phidgets API

Phidgets Inc. is a Canadian company which produces different sensors and controllers. `phidgetapi` is an API package written in JavaScript developed by Brandon Nozaki Miller. It supports many Phidgets sensors. This module is compatible with different OSs: Linux, Mac OS X, and Windows, which can run `node.js` or `io.js` [10].

There are 6 different libraries supporting 8 types of sensor, including: GPS, RFID, Servo, Spatial, Analog, Motor Control, Weight and Temperature.

API

- Each sensor type has their own set of methods and data which are provided to access and control that sensor. Although each sensor type has different API to work with, but all of them maintain a similar scheme which is easier for developers to use. To be more specific, each library provides the same interface for

connecting/disconnecting sensor, observing/unobserving sensor changes, and registering function to be run when it is ready.

Method	Input	Return value	Description
connect	Object phidget.params	None	Connects to the sensor
quit	None	None	Disconnect from the sensor
whenReady	Function callback	None	This callback function will be executed when the sensor is ready to be used
observe	Function callback	None	Used for asynchronously observing the changes of the sensor
unobserve	Function callback	None	Stop observing from the specified observe change handler function

Table 1. *Phidgets API - Basic methods of all sensor type*

- If there are more than one properties need to be observed, multiple observing methods are provided. For example, in the case of Temperature module, there are two different properties need to be observed: sensor temperature and ambient temperature. Therefore, Temperature library delivered 4 different observing methods as follows:

Method	Input	Return value	Description
observeTemperature	Function callback	None	Used for asynchronously observing the changes of the temperature of the sen-
unobserveTemperature	Function callback	None	Stops observing temperature of the
observeAmbientTempera-ture	Function callback	None	Used for asynchronously observing the changes of the ambient temperature
unobserveAmbientTem-perature	Function callback	None	Stops observing ambient temperature

Table 2. *Phidgets API - Observe/Unobserve methods of the Temperature module*

- Depending on specific sensor type's features, additional methods can be presented to support developers if needed. However, not all of sensor types need to implement additional functions. In the case of Temperature module, no additional functions is presented. Meanwhile, there are three additional functions provided by the Spatial module:

Method	Input	Return value	Description
zeroGyro	None	None	Re-zeroes the gyroscope.

setCompassCorrectionParameters	Number magfield, Number offset0, Number offset1, Number offset2, Number gain0, Number gain1, Number gain2, Number T0, Number T1, Number	None	This function adjusts the parameters of the compass.
resetCompassCorrectionParameters	None	None	Resets the Compass Correction Parameters to default values:(1,0,0,0,1,1,1,0,0,0,0,0,0)

Table 3. *Phidgets API - Additional methods in the Spatial module*

- Moreover, developers can access and retrieve sensor values as well as sensor attributes by using set of data provided by the libraries. For example, the Temperature library has a set of data as follows:

Data	Data type	Writable	Description
type	String	no	“PhidgetTemperature”
ambientTemperature	Number	no	Ambient temperature in Celsius
ambientTemperatureMax	Number	no	Max ambient temperature in Celsius
ambientTemperatureMin	Number	no	Min ambient temperature in Celsius
temperature	Number	no	Sensor temperature in Celsius
temperatureMax	Number	no	Max sensor temperature in Celsius
temperatureMin	Number	no	Min sensor temperature in Celsius
potential	Number	no	Thermocouple potential in Millivolts
potentialMax	Number	no	Max thermocouple potential in Millivolts
potentialMin	Number	no	Min thermocouple potential in Millivolts
thermocoupleType	Number	no	Thermocouple key

Table 4. *Phidgets API - Set of data in the Temperature module*

- Events: The core `Phidget` module provides events that can be handled by each sensor type module as well as by developers.

Event name	Description

phidgetReady	The phidget is attached and fully initialized
error	Emit whenever phidget have an error
changed	Emit whenever phidget or sensor has data which has changed
attached	Phidget attached to computer (connect via USB cable)
detached	Phidget detached from computer (connect via USB cable)
log	When `rawLog` is set to true, this event will be fired as data comes over the raw phidget socket.
disconnected	The phidget socket was closed or lost (connect via Ethernet cable)
connected	The phidget socket was found and connected (connect via Ethernet cable)
data	other Phidget data such as: management or configuration data

Table 5. *Phidgets API - Events*

Implementation

- All sensor classes inherit from a base class named `Phidget`. This class is extended from `EventEmitter` class of Node.js. The `Phidget` class is responsible for tasks requiring interaction with lower layer such as socket connection, sending data package through socket. This core class also provides methods used for binding/unbinding event listener as well as emits basic events (see table 5) that are handled by each sensor modules.

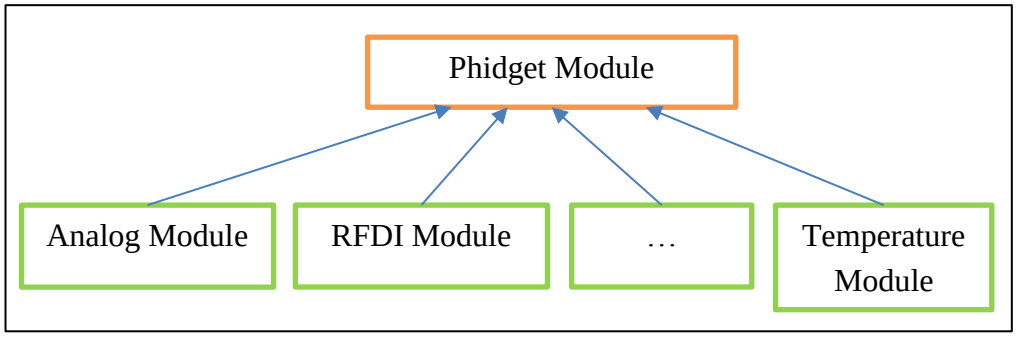

Figure 1. *Phidgets API - Inheritance structure*

- Each sensor module inherits from the core `Phidget` module. It also defines its own methods and data. Besides, some of events emitted from the Phidget class are handled in advance and wrapped into methods in each sensor module. The most common pre-handled events in sensor modules are: `whenReady`, `log`, `data`, `changed`, and `error`. As the result of that, in most cases, developers do not need to use `'on'` method of the `EventEmitter` class to handle events but using methods provided by its module instead.

- The input function of `whenReady()` method will be executed only one time as default. Therefore, if developers want to automatically trigger it after a specific time, they should set interval for this event. There is no method handling it automatically.

Example

```
var Phidget = require('phidgetapi').TemperatureSensor;
var temp = new Phidget;

temp.connect();
temp.whenReady(init);

function init(){
      setInterval(
            function(){
                  temp.observeAmbientTemperature(ambientTemperatureUpdates);
                  temp.observeTemperature(temperatureUpdates);
            }, 5000);
}

function ambientTemperatureUpdates(changes){
    console.log('Ambient Temperatures', temp.ambientTemperature);
}

function temperatureUpdates(changes){
    console.log('Sensor Temperatures' , temp.temperature);
}
```

Program 1. *Phidgets API – A sample code*

3.1.2 Tessel API

Tessel is a company which provides internet-enabled devices for software developers. Tessel is also the name of microcontrollers (Tessel & Tessel 2) developed by this company. Tessel microcontrollers can connect to Internet and are programmable in JavaScript, which helps developers extend the reach of the Web to physical devices. Tessel APIs are open source code developed by Tessel's team that uniquely support Tessel's modules. The code is written in JavaScript and can be installed through npm (Node.js Package Manager).

Regarding libraries for sensors, Tessel supports: Accelerometer (for 'accel-mma84' Module), Climate (to read Humidity and Temperature from 'climate-si7020' Module), Ambient (to read sound and light from 'ambient-attx4' Module), GPS (for 'gps-a2235h' Module), Infrared (for 'ir-attx4' Module), Relay (for 'relay-mono' Module), RFID (for 'rfid-pn532' Module), and Servo (for 'servo-pca9685' Module) [11]. More libraries can be added in the future.

API

- Each sensor library was implemented separately. Each module provides its own set of methods that are necessary to interact with its sensor, such as for configuration and reading data, etc. So, instead of using list of data like Phidgets, users control and get necessary data from sensors through defined methods. There are no common rules in naming methods. Generally, most of methods provide a callback function as their parameter, so it can be called after the task is completed.
- However, every library module provides a one common method named `use`, which are used to initialize the connection between the sensor module and the main board. Each type of module can only run on specific port and using particular peripheral protocol, such as I2C, SPI, UART, etc. Therefore, users need to define the name of port their sensor connected with as the input parameter of `use` method. Each sensor module has their own instructions posted on the Tessel websites. Information about the port can be found there. For example:

```
var tessel = require('tessel');
var climatelib = require('climate-si7020');
var climate = climatelib.use(tessel.port['A']);
```

Methods below are those provided by the Climate module (for reading temperature and humidity of ambient environment):

Method	Input	Return value	Description
readTemperature	String format, Function callback	None	Read the temperature in Celsius or Fahrenheit. The `callback` function will get errors or temperature as arguments. `format` parameter indicates the temperature unit, it should be `'c'` or `'f'`
readHumidity	Function callback	None	Read the humidity
setHeater	Boolean status, Function callback	None	Set the HEAT(*) configuration register
use	Object hardware	Object device	Initialize the connection between sensor module and the main board. `hardware` parameter indicates the port that the module connected

(*) The heater helps to increase the accuracy of humidity measurement. However, it changes ambient temperature, thus affects on temperature measurement.

Table 6. *Tessel API – Methods in the Climate module*

- Events: There are two basic events which every sensor module emitted, including: `'ready'` and `'error'`. Other events are emitted according to its demand. Names of events are different between modules.

Event name	Description
ready	Emitted when the connection between the Tessel and the Temperature module is successful.
error	Emitted when an error occurs

Table 7. *Tessel API - Events emitted by the Climate module*

- According to the Tessel API design, sensor data can be read in different ways:
    a. Reading once by calling reading method directly.
    b. Reading periodically by setting interval for the reading method.
    c. Reading data through an event provided from the module.

  It is worth noting here that not every sensor module provides an event whenever new data is available, such as the Climate module. Moreover, there is no common name for such event. In most of modules, it might be named as `'data'` event, but in some cases it is changed, such as in the Ambient module. Because of there are two properties measured at the same time by this module, thus two different events are offered: `'sound'` and `'light'`.
- Triggering events based on threshold value can be found in the implementation of some modules, such as in the Ambient module: `'light-trigger'` and `'sound-trigger'` are corresponding to events when light and sound value pass threshold restrictions.

Implementation

- Unlike Phidgets API, in Tessel API, each sensor object inherits directly from `EventEmitter` class. Therefore, all tasks are handled by each sensor object. Because each sensor board uses different protocols to connect and transmit data, it is difficult to handle all connection tasks in one class like Phidgets does.
- Tessel hardware API was implemented separately from the sensor modules. This library is responsible for managing hardware parts, such as: pins, ports, and LEDs. Developers must include the `'tessel'` library in the beginning of their code. However, developers do not have to concern much about it except using its data to provide the appropriate port for the sensor module.
- All internal methods which are used to access to hardware layers and establish the communication channel (through UART, I2C, SPI protocols, etc.) to the sensor module are named with the underscore prefix. Users should not use those methods in their code.
- The main difference between Tessel API and Phidgets API is that applications acquire needed data through methods rather than through data members.

Example

```
var tessel = require('tessel');
var climatelib = require('climate-si7020');
var climate = climatelib.use(tessel.port['A']);

climate.on('ready', function(){
  setInterval(function(){
    climate.readHumidity(function(err, humid){
      climate.readTemperature('f', function(err, temp){
        console.log('Degrees:', temp + 'F');
        console.log('Humidity:', humid + '%RH');
      });
    });
  }, 1000);
});

climate.on('error', function(err) {
  console.log('error connecting module', err);
});
```

Program 2. *Tessel API – A sample code*

3.1.3 Insteon Hub API

Insteon is a company that produces home automation products. Its products are based on Insteon technology which allows different home devices interoperate through power lines, radio frequency, or Internet. Insteon Hub Sensor API is a node package to control home automation devices written by Brandon Goode. The API uses the direct Power-Linc Modem (PLM) connection over Transmission Control Protocol (TCP), Serial connection or the cloud [12].

It provides libraries for: Door sensor (Open/Close), Leak sensor (notify water leak), Light sensor (On/Off), Motion sensor (monitor motion), Meter sensor (monitor power consumption), Thermostat sensor (Temperature & Humidity), IO and Actuator sensor [13].

API

- `Insteon` base module is responsible for the connection, linking, and sending commands. Most of methods in `Insteon` module are used to support the implementation of other sensor modules. However, connection and linking tasks may be necessary for users to create a connection to the gateway or to a Power-Linc modem, link/unlink device(s) to gateway, etc. In that cases, appropriate methods of `Insteon` can be used in applications also.
- At the beginning of an application, a sensor object needs to be created by calling appropriate method in `Insteon` class. Generally, that method has the same name as the sensor. The id input parameter is the identification string of the sensor that set by applications. For example, Insteon class provides these initializing methods to create sensor objects as follows:

Method	Input	Return value	Description
light	String id	Object device	Create a lighting object
thermostat	String id	Object device	Create a thermostat object
motion	String id	Object device	Create a motion object
meter	String id	Object device	Create a meter object
door	String id	Object device	Create a door object
leak	String id	Object device	Create a leak object
io	String id	Object device	Create a io object

Table 8. *Insteon Hub API - Initializing methods in the Insteon module*

- Insteon Hub API aims to develop applications of home automation, thus they provide not only methods to access and retrieve data from sensors but also methods to control devices. These methods can be called through the object which is returned by calling above initializing methods. Most of sensor methods support callback function as its parameter. But from the version 4.0 afterward, all functions return promises. The callback function is now optional. For example:

Method	Input	Return value	Description
tempUp	Number change, [Function callback]	Promise next	Increase the temperature. `change` parameter indicates the increased degree, default value is 1.
tempDown	Number change, [Function callback]	Promise next	Decrease the temperature. `change` parameter indicates the decreased degree, default value is 1.
temp	Number zone, [Function callback]	Promise next	Get the current air temperature of a specific zone. Default zone is 0.
coolTemp	Number temp, [Function callback]	Promise next	Set the cool temperature.
heatTemp	Number temp, [Function callback]	Promise next	Set the heat temperature.
highHumidity	Number level, [Function callback]	Promise next	Set the high humidity level (1 to 100)
lowHumidity	Number level, [Function callback]	Promise next	Set the low humidity level (1 to 100)
status	[Function callback]	Object details / Promise next	Get the status of the themostate.

(*) Ramp rate is the rate of how fast the light turn on or off

Table 9. *Insteon Hub API – Some methods in the Thermostat module*

- Besides, each module also provides necessary events that are suitable to each sensor type. In addition, most of sensor module provides `'heartbeat'` event which is emitted every 24 hours by the sensor to inform users whether it is alive.

Event name	Description
heartbeat	Emitted every 24 hours
cooling	Emitted when the thermostat starts cooling
heating	Emitted when the thermostat starts heating
off	Emitted when the thermostat stop heating or cooling (ie. System is off)
highHumidity	Emitted when humidity is above the high humidity set point
lowHumidity	Emitted when humidity is below the low humidity set point
normalHumidity	Emitted when humidity returns to normal levels

Table 10. *Insteon Hub API - Events provided by Thermostat module*

Implementation

- As same as Phidgets API and Tessel API, in the Insteon Hub API, each library was implemented separately in different file.
- Both `Insteon` and sensor modules inherit from `EventEmitter` class of Node.js.
- Basically, Insteon APIs is implemented as same as Tessel's APIs. It means they both provides basic events for user to bind their own handler function, and other methods to control and read different data from sensors. However, in lower level, because Tessel connects with sensors via peripheral ports while Insteon communicates with sensors via TCP/Serial/Cloud, so the lower-level implementation are different.

Example

```
var hub = new Insteon();
var thermostat = hub.thermostat('112233'); /* '112233' is the id of the ther-
mostate object */

thermostat.status(function(data) {
      console.log('Current temperature: ' + data.temperature + ' ' + da-
ta.unit + ' degree');
});

thermostat.on('cooling', function() {
      console.log('Temperature is decreasing');
      // increase it
```

```
      thermostat.tempUp(function() {
            console.log('Temperature has increased one degree');
      });
});
```

Program 3. *Insteon Hub API - A sample code*

3.2 Sensor APIs on native platforms

Although the target of my thesis is to design and implement a Sensor framework API in JavaScript, but I found that Sensor APIs which were developed on native platforms such as Android and iOS are worth doing research. The reason here is they provide full-fledged frameworks which are highly standardized and structured. These frameworks gave me a perspective on building a framework with multi-layers, which is convenient in managing and maintaining the system. Because these Sensor APIs were not implemented by JavaScript, then I ignore features of their implementation but focus on their APIs instead.

3.2.1 Android Sensor Framework

Android provides generic API for sensor since API level 3 and has not changed much after introduction. This framework was written in Java. Currently, Android supports 13 types of sensors, divided into 3 categories: Motion sensor (accelerometers, gravity sensors, gyroscopes, and rotational vector sensors), Environmental sensors (barometers, photometers, and thermometers), and Position sensors (orientation sensors and magnetometers). GPS is not included in Position sensors, and it will be handled in different way [14].

API

Android Sensor Framework has 4 classes: `Sensor Manager`, `Sensor`, `SensorEvent`, and `SensorEventListener`.

- `SensorManager`: to create an instance of the sensor service, allowing user to access the device's sensors, register/unregister sensor event listeners, provide batch mode (API level 9), etc. It contains:

    a. Public methods: `SensorManager` class provides public methods that are used for:

        i. Accessing and listing sensors

Method	Input	Return value	Description
getDefaultSensor	int type, [boolean wakeUp]	Sensor	Get a default sensor for a given type (and wakeUp property if it is available)

getSensorList	int type	List<Sensor>	Get list of available sensors of a certain type

Table 11. *Android API - Accessing and listing sensors methods*

ii. Managing sensor event listeners and trigger events

Method	Input	Return value	Description
cancelTrigger-Sensor	TriggerEventListener listener, Sensor sensor	boolean	Cancel receiving trigger events for a trigger sensor
requestTrigger-Sensor	TriggerEventListener listener	boolean	Request receiving trigger event for a trigger sensor
registerListener	SensorEventListener listener, Sensor sensor, [optional params…]	boolean	Register a listener for a given sensor (with certain properties)
unregisterListener	SensorEventListener listener, [Sensor sensor]	void	unregister a listener (for all sensors or for a given sensor)

Table 12. *Android API - Methods handling triggers and listeners in the SensorManager class*

iii. Acquiring orientation information, such as: computing the device's rotation, computing the geomagnetic inclination angle, converting a rotation vector to a rotation matrix, etc.

b. Constants: representing values used in measuring gravity, light, magnetic fields, pressure, etc. It supports:
   i. 4 delay modes. They are used as an input parameter of the `registerListener()` function (read more from the table 12 or the program 4).

Constant	Value	Description
SENSOR_DELAY_FASTEST	0	Get sensor data as fast as possible
SENSOR_DELAY_GAME	1	Rate suitable for games
SENSOR_DELAY_UI	2	Rate suitable for the user interface
SENSOR_DELAY_NORMAL	3	Rate(default) suitable for screen orientation changes

Table 13. *Android API - Delay modes*

ii. 3 accuracy modes

Constant	Value	Description
SENSOR_STATUS_ACCURACY_LOW	1	Low accuracy, calibration with the environment is needed

SENSOR_STATUS_ACCURACY_MEDIUM	2	Medium accuracy, calibration with the environment may improve the readings
SENSOR_STATUS_ACCURACY_HIGH	3	Maximum accuracy

Table 14. *Android API - Accuracy modes*

iii. 2 sensor statuses

Constant	Value	Description
SENSOR_STATUS_NO_CONTACT	-1	The values returned by this sensor cannot be trusted because the sensor had no contact with what it was measuring
SENSOR_STATUS_UNRELIABLE	0	The values returned by this sensor cannot be trusted, calibration is needed or the environment does not allow readings

Table 15. *Android API - Status modes*

- `Sensor`: to create an instance of a specific sensor, get basic info of sensors (range, delay, power, resolution, vendor, version, etc.). It contains:
    a. Constants: represent reporting mode and describe different sensor types. It supports:
        i. 4 reporting modes

Constant	Value	Description
REPORTING_MODE_CONTINUOUS	0	Events are reported at a constant rate
REPORTING_MODE_ON_CHANGE	1	Events are reported when the value changes
REPORTING_MODE_ONE_SHOT	2	Events are reported in one-shot mode
REPORT-ING_MODE_SPECIAL_TRIGGER	3	Events are reported as described in the de-scription of the sensor

Table 16. *Android API - Constants representing reporting modes in the Sensor class*

ii. 21 types of sensor. The constants representing sensor types are either in string or number format.
For example:

Constant	Value	escription
TYPE_ALL	-1	Describing all sensor types
TYPE_LIGHT	5	Describing an light sensor type
TYPE_GRAVITY	9	Describing a gravity sensor type
STRING_TYPE_LIGHT	“android.sensor.light”	String describing an light sensor type
STRING_TYPE_GRAVITY	“android.sensor.gravity”	String describing a gravity sensor type

Table 17. *Android API – Some constants representing sensor types in the Sensor class*

b. Public methods: are used to determine a sensor's information, such as: name, type, vendor, power, reporting mode, etc.
For example:

Method	Input	Return value	Description
getName	void	string	Get name of the sensor
getPower	void	float	Get power in mA used by the sensor

Table 18. *Android API - Some public methods in the Sensor class*

- `SensorEvent`: to create a sensor event object, provide information about a sensor event. These are information that a sensor event object provides:

Field	Data type	Description
accuracy	public int	The accuracy of the event
sensor	public Sensor	The sensor that generated the event
timestamp	public long	The time when the event happened
values	public final float[]	Event value

Table 19. *Android API - Information provided by the sensor event object*

Here, Android uses an amorphous array of floats to store all sensor data types. In addition, the length of the `values` array is varied between different sensor types.

- `SensorEventListener`: supports 2 callback methods corresponding to: sensor value changes, and sensor accuracy changes. These two callback methods are used to monitor raw sensor data.

Method	Input	Return value	Description
onAccuracyChanged	Sensor sensor, int accuracy	abstract void	Called when the accuracy of the registered sensor has changed
onSensorChanged	SensorEvent event	abstract void	Called when a sensor report a new value

Table 20. *Android API - Callback methods in the SensorEventListener class*

Example

```
public class SensorActivity extends Activity implements SensorEventListener {
  private SensorManager mSensorManager;
  private Sensor mPressure;
```

```
  @Override
  public final void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    // Get an instance of the sensor service, and use that to get an instance
    // of a particular sensor.
    mSensorManager =(SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mPressure = mSensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE);
  }

  @Override
  public final void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Do something here if sensor accuracy changes.
  }

  @Override
  public final void onSensorChanged(SensorEvent event) {
    float millibars_of_pressure = event.values[0];
    // Do something with this sensor data.
  }

  @Override
  protected void onResume() {
    // Register a listener for the sensor.
    super.onResume();
    mSensorManager.registerListener
(this,mPressure,SensorManager.SENSOR_DELAY_NORMAL);
  }

  @Override
  protected void onPause() {
    // Be sure to unregister the sensor when the activity pauses.
    super.onPause();
    mSensorManager.unregisterListener(this);
  }
}
```

Program 4. *Android API – A sample code*

3.2.2 iOS Core Motion Framework

In fact, iOS does not have a generic sensor framework API like Android, but provides specific APIs for each type of sensor. However, from iOS 5 version, iOS provides some kind of a unified framework for motion sensors, called "Core Motion Framework". This framework were written in Object-C language. It supports: accelerometer, magnetometer, and gyroscope, and other device motion sensors [15]. CoreMotion API is only generic for motion sensors. Other type of non-motion sensors are excluded.

API

- Most of functionalities of the framework are encapsulated in the class `CMMotionManager`.

- In general, each motion module provides: an interval property, an update method, a handler object, and a data event object.
  For example, the Gyroscope module provides:

Name	Type	Description
gyroUpdateInterval	property	Specify the update interval
startGyroUpdates	method	Update data with a update handler
CMGryoHandler	object	The type of block callback for handling gyroscope data
CMGyroData	object	Contains a single measurement of the device's rotation rate

Table 21. *iOS Core Motion API - Elements supported by the Gyroscope module*

- There are two ways for application to receive motion data:
  a. Handling the motion updates at specified intervals by:
     i. Setting the update interval by using: `xxxUpdateInterval()` function (with `xxx` is the name of sensor type, including: `accelerometer`-, `gyro`-, `magnetometer`-, and `deviceMotion`-).
     ii. Calling the update method: `startXxxUpdates`, and passing in the callback `CMXxxHandler`
     iii. Retrieving data through the `CMXxxData` object.
  b. Periodic sampling of motion data by:
     i. Calling the update method: `startXxxUpdates` without the callback
     ii. Retrieving `CMXxxData` object by reading `xxxData` property.
- iOS Core Motion framework supports different classes representing a motion event, which inherits from the class `CMLogItem` (represents a piece of time-tagged data that can be logged to a file). For example: `CMAccelerometerData`, `CMAltitudeData`, `CMGyroData`, `CMMagnetometerData`, etc. Through these data event object, users can access the motion data by reading their data properties, including: `CMAcceleration`, `CMAltitude`, `CMMagneticField`. Unlike Android's approach, iOS Core Motion framework predefine different structures for different motion data.
  For example: `CMAccelerometer` has the following structure:

Property	Data Type	Description
x	Double	X-axis acceleration in G's (gravitational force)
y	Double	Y-axis acceleration in G's

z	Double	Z-axis acceleration in G's

Table 22. *iOS Core Motion API - Properties of CMAccelerometer class*

While the `CMAltitude` class provides:

Property	Data Type	Description
relativeAltitude	NSNumber	The change in altitude (meters) since the last reported event
pressure	NSNumber	The recorded pressure, in kilopascal.

Table 23. *iOS Core Motion API - Properties of CMAltitude class*

- `CoreMotion` provides two modes of accessing device motion samples: push and pull. The received (push) or retrieved (pull) device motion samples are encoded into an instance of `CMDeviceMotion` which encapsulates one sample of device motion data at a given time.

Example

```
CMMotionManager *motionManager;
CMAttitude *referenceAttitude;
motionManager = [[CMMotionManager alloc] init];
referenceAttitude = nil;
-(void) enableMotion{
        CMDeviceMotion *deviceMotion = motionManager.deviceMotion;
        CMAttitude *attitude = deviceMotion.attitude;
        referenceAttitude = [attitude retain];
        [motionManager startDeviceMotionUpdates];
}
```

Program 5. *iOS Core Motion API – A sample code*

3.3 W3C Generic Sensor API Specification

W3C (World Wide Web Consortium) is an international community which builds standards including protocols and guidelines for the Web. It was found in 1994 October by Tim Berners-Lee – the inventor of the World Wide Web. The aim of W3C is to build standards which help Web applications to be more consistent, stable, secure, and effective [23].

The W3C Generic Sensor API Specification is a project conducted by W3C Device and Sensor Working Group. This project was started in 2015 with an aim is to create a specification for exposing sensor data to the Open Web Platform in a consistent way. With the fast development of IoT, this specification will create a convenient environment for developers to develop sensor-related Web application based on W3C specification's blueprint. The specification will provide a sensor interface that is flexible enough to

accommodate different sensor types. It is seen as the most important movement toward standardization generic sensor APIs so far. Although this project is still a work-in-progress and have a long journey ahead to become a W3C Recommendation, it demonstrates the important of standardizing the way sensors are exposed on Web environment. Its result will propel the development of IoT in more consistent way. Therefore, going through its drafts sheds light on some basic ideas of implementing a sensor framework. Therefore, I included my research on this specification in my thesis, despite the fact that it is a general specification without actual implementation.

One more point is worth noting here is that examples using in the W3C specifications were written in a formal language which is independent of any specific programing languages. However, to be more convenient, I will present pieces of code in JavaScript instead.

Scope

This specification limitedly covers local sensors rather than expanding to remote ones (neither remotely connected nor be found in personal area network). Besides, the sensor discovery API is also out-of-scope. Instead of that, it proposes a strategy called "Feature Detection of Hardware Features" as an alternative solution.

API

- This Working Draft proposed an unfledged `Sensor` interface. The `Sensor` class is extended from the `EventTarget` class and provides some basic attributes, methods, and events as follows:

Attribute	Data Type	Writable	Description
state	SensorState	No	State of the sensor
reading	SensorReading	No	Holding the time value

Table 24. *W3C Generic Sensor API - Attributes of the Sensor class*

Method	Input	Return value	Description
start	void	void	Activating the sensor for being ready to use
stop	void	void	Deactivating the sensor

Table 25. *W3C Generic Sensor API - Sensor methods*

Event	Description
onchange	Emitted when data is change

onstatechange	Emitted when sensor state is change
onerror	Emitted when an error occurs

Table 26. *W3C Generic Sensor API - Events*

- There are four states defined by the specification, including:

State	Description
“idle”	Idle state, when the sensor is deactivated or stopped
“activating”	Activating state, when the sensor is waiting for registering and updating reading in the first time
“activate”	Activate state, when the sensor is registered and ready to use
“errored”	Errored state, when the sensor cannot work properly

Table 27. *W3C Generic Sensor API - States*

- `Sensor` is a base class for all concrete sensor classes. Hence, Sensor class only provides very common and basic methods, events, and attributes that are needed in most of sensor types. Each specific sensor type such as ambient light, temperature, pressure, etc. needs to implement its own class which is extended from the `Sensor` base class and adds their own parts as well. Events, states, and `start()`, `stop()` methods are supported by all sensor types.
  For example, if Ambient Light sensor class is already implemented, then users can create its object and access to it by using pre-defined methods and events like this:

```
var sensor = new AmbientLightSensor();
sensor.start();

sensor.on('onchange', function(event) {
    console.log(event.reading.illuminance);
};

sensor.on('onerror', function(event) {
    console.log(event.error.name, event.error.message);
};
```

Program 6. *W3C Generic Sensor API – A sample code*

- Besides `Sensor` interface, other associated interfaces are also tentatively suggested (they possibly can be changed in the future if needed), such as: `SensorReading`, `SensorReadingEvent`, and `SensorErrorEvent`:
  a. `SensorReading`: an instance of this class represent a reading of a sensor at a given time. It contain a `DOMHighRestTimeStamp` attribute holding the time when the reading occurs.

b. `SensorReadingEvent`: is extended from the DOM `Event` class. (DOM – Document Object Model is a W3C standard which is used to access a document). An instance of this class represent an event of a sensor at a given time. (see the program 6)
c. `SensorErrorEvent`: is extended from the DOM `Event` class. An instance of this class represents an error event. (see the program 6)

These interfaces have pre-defined attributes as bellows:

Attribute	Interface	Attribute type	Description
timestamp	SensorReading	DOMHighRes-TimeStamp	A timestamp at the time a reading was obtained from a sensor
reading	SensorReadingEvent	SensorReading	A reading of a sensor at the time an event occurs
error	SensorErrorEvent	Error	An error event

Table 28. *W3C Generic Sensor API - Pre-defined attributes of SensorReading, SensorReadingEvent, and SensorErrorEvent interfaces*

- Because the reading of each specific sensor type depends on its own data structure. Therefore, along with the `timestamp` inherited from the `SensorReading` class, each sensor type defines another attribute to hold its sensor data. For example, an `AmbientLightSensorReading` class is defined as follows:

```
const util = require('util');
var AmbientLightSensorReading = function(illuminance) {
      this.illuminance = illuminance; //light level
}
util.inherits(AmbientLightSensorReading, SensorReading);
```

As the result of that, the current light level can be achieved by observing the event `'onchange'` and by reading `illuminance` attribute from the reading event as follows:

```
sensor.on('onchange', function(event) {
    console.log(event.reading.illuminance);
};
```

- Reporting mode:
  a. Periodic: when sensor is read after a specific interval.
  b. Auto: the sensor is triggered when there is a measurable change.

  However, the reporting mode also depends on the underlying implementation of how sensor readings is acquired.

Implementation

- The draft provides detailed instructions for:
    a. Constructing a sensor object
    b. Registering a sensor
    c. Unregistering a sensor
    d. Setting sensor settings
    e. Observing a sensor
    f. Finding current Reporting mode of a sensor
    g. Finding the polling frequency of a sensor
    h. Updating state
    i. Updating current reading
    j. Updating reading
    k. Handling errors
    l. Requesting sensor access
- The specification assumes that devices using more than one sensor of a specific type are rare. Therefore, it suggests that the API should be implemented in the way that is easy for developer to interact with the default sensor of each type (as same as Android's strategy). In the case if it is difficult to distinguish and define a default sensor among them, more specific information should be passed as extended parameters when instantiating. For example:

```
var sensor = new DirectTirePressureSensor({position: "rear", side: "left"});
sensor.on('onchange', function() {
    console.log(event.reading.pressure);
});
```

- Feature Detection of Hardware Features: Along with strategy of checking availability of needed sensor API, the specification suggests additional strategies to protect application from unexpected interruptions when accessing to the sensor. According to it, programs should check and listen every errors emitted from the sensor and handle it in advance to enhance user's experience without degrading it.
- Privacy and security issues are also considered in the draft with general suggestions such as: sensor readings must only be available in the top-level browsing context and in security context; using Permission API to handle process of accessing sensor readings.
- Along with defining the Sensor interface, the specification also gave instructions of how basic functions such as `start()`, `stop()` should be implemented.
- The Working Draft has a separate section for extensibility in which includes instructions of how to extend to specify APIs for different sensor types. Nevertheless, there are just only simple rules for naming convention, units, defining a default, etc. Tons of work need to be done in the future.

Discussions

The Device and Sensor Working Group and other contributors have been openly discussing about matters relating to the Generic Sensor API Specification on its GitHub site [16]. I noted some major questions and debates surrounding this topic that are worth considering.

- Does it need to provide a way of tying sensor request to animation frames? Some developers raised the question about getting updated sensor value per animation frame. It is necessary in the game and other animation frame loop-based situation. Therefore, a setting like

  ```
  frequency: “animationframe”
  ```

  is suitable for such cases. The Working Group agreed that this feature is desirable to consider and requested further research to see whether or not that is implementable.
- How to implement one-shot readings? Besides getting sensor readings through `ondata` and `onchange` triggers. In some use cases, developers need to get unique data point from a sensor (for example, achieve current user’s position through Geolocation sensor). Moreover, if developers care about battery life, they also want to get sensor value only if it’s cached. Although, the `read()`, and `readFromCache()` can be used in such situations, but the point is that developers usually do not want to observe the sensor in such those cases. Therefore, the WD suggested 2 solutions for this issue:
  a. Sensor should not start polling automatically after instantiated. Instead of that, a function would be provided to activate the polling operation.
  b. While sensors still start automatically polling when instantiated, the API provides a class method for one-shot reading.

  Both solutions have their own pros and cons, and the decision is still on the fence.
- How to implement data batching? Data batching is needed in case developers have to deal with high data frequency. In this situation, the polling frequency is smaller than the sensor frequency. Therefore, there is batch of sensor value gathered after one trigger. So the question here is how API can be implemented to handle data batching. Currently, WD still cannot make any decision about this issue. Further research need to be carried to assess related requirements such as performance and memory constraints.
- How should Generic Sensor API define sensors’ unit option? This issue started from the fact that many different types of unit representing the same measured property. The typical example is that temperature can be measured in Celsius, Fahrenheit, or Kelvin degree. More complicated cases can be mentioned as Proximity (cm – centimeter, in – inch), Tait-Bryan angles (x-y-z, y-z-x, z-x-y, x-z-y, z-y-x, y-x-z) for device orientation, etc. At this moment, WD suggests that

the default unit should be defined in each implementation of the specification. Simultaneously, an error should be emitted if the sensor does not support a required unit.

- Some enhanced features (for the next version) are presented in the GitHub discussion site, such as: encrypted data strategy, sensor's background behavior, etc.
- Lastly, there are quite many questions relating to implementation details, such as:
    a. Should `SensorReading` carry reading payload?
    b. Should the enum `SensorState` have `'nosensor'` attribute?
    c. Does `SensorReading` always need a `TimeStamp`?
    d. Constructing a `SensorReading` with `SensorReadingInit` is non-optional?

To conclude, quite a large number of issues have being posted and discussed among specialists and developers. Many of them are still unsolved or require deeper research. Thorny problems usually relate to security and discovery strategies. Besides, I believe more and more topics will be raised in the future paralleling to maturity of the specification. However, it is a good sign because it demonstrates that international communities have recognized the important of this project and they are implementing it in a very careful manner.

3.4 Comparison of different approaches

Hardly to say there are any approaches that I've presented have all bad or good sides. In this section, I compare these Sensor APIs by picking up different approaches and implementation styles and present their advantages and drawbacks from different angles.

3.4.1 Suitability

Suitability of a framework determines how that framework is suitable in different platforms or sensor types.

Among different approaches, Android framework allows users to handle different sensor configuration for its devices. Sensor configuration determines what sensors are included in a device. Without a standard sensor configuration, manufacturers are free to choose what sensors they want to include, and what they do not [14]. On the other hand, iOS CoreMotion supports only motion sensors such as: accelerometer, magnetometer, and gyroscope. Phidget, Tessel's, and Insteon Hub APIs supports only their own brand sensors and modules. Nevertheless, these framework structure can be extended to support other sensor types if needed. For example, in case of iOS CoreMotion framework, new module only needs to provide its own interval properties and an update method with a handler object as its input parameter. Besides, it is necessary to define a class

representing that module's data structure. After that, users can access to the new sensor by using the same strategy they do with other existing sensors.

Regarding to running platforms, iOS CoreMotion and Android sensor frameworks are developed based on their native platforms, thus they are unfit on Open Web environment. On the other hand, Phidget, Tessel, and Insteon Hub are written in JavaScript, which is suitable to different OSs which can run Node.js or io.js. It means sensors can be exposed on Web platform, which allows users to access sensor data through Web pages. Besides, although current W3C's design based on `EventTarget` but they have plan to switch to `EventEmitter` to avoid burden of dependency on DOM and fit to non-browser platforms. Recall that JavaScript API is the goal of this thesis.

3.4.2 Flexibility

In general, constructing a generic class and allowing each specific sensor to extend functionalities from it provides a more concrete and consistent structure than implementing them individually. Although all approaches I have conducted researches use this strategies, only Android Sensor Framework and W3C Generic Sensor API Specification provide a comprehensive multi-layer design. To be more specific, while Phidget, Tessel, and Insteon Hub API all have a base class, that class mostly plays a role as a middle layer which handles tasks relating to lower layer such as socket connection, sending/receiving package of data by using different hardware protocols (UART, SPI, I2C, etc.), or connecting to Cloud services, etc., rather than plays a role as a generic class which provides basic sensor data, methods, and events. On the other hand, Android and W3C specification provides that architecture, which is considered to be more flexible.

Next, because JavaScript is a dynamically and loosely typed language, all JavaScript APIs representing above use a generic approach for storing sensor type. JavaScript APIs use an object or an array-based variable for all sensor types. The advantage of that generic approach is that developers do not have to wait for a data standard every time a new sensor is introduced. However, it requires a specific specification to explain the structure of each sensor data type representing in array for parsing data. As a result, from application developers' perspective, it seems this approach is overly generic. It would be nice for contextually relevant property name of sensor data rather working on generic one. At the other side, although building each sensor data structure brings more comfortable zone for developers, but it is more complicated when adding new sensors into the framework.

Regarding to sensor discovery issue, there are no perfect solution to cope with this problem. Although Android provides a mechanism to detect sensor features at run-time but it is unable to apply to JavaScript frameworks. It is because Android sensor framework usually works with local and fixed sensors attached on the device. Thus, feature detec-

tion is done upfront and does not affect performance. Regarding to JavaScript sensor frameworks, they need to get some data out of the sensor to confirm its aliveness. Nevertheless, it is costly because getting hardware information is time consuming and battery draining. Insteon Hub confronts with this problem by emitting event heartbeat every 24 hours to confirm that "I'm still alive". Meanwhile, W3C suggests quite the same strategy but using writable `timeout` values (with default one) for each type of sensor.

Last but not least, most of sensor libraries do not support multiple instances of the same sensor type. However, Android has. It allows user to list all the sensors of a given types, and using `getDefaultSensor()` to get the default sensor assigned for each type. However, in some cases assigning default sensor is inappropriate, e.g. proximity sensors of a car, user should be allow to choose not to define a default sensor. It is a more flexible approach and W3C has the same view on this issue.

In brief, although every frameworks have pros and cons, they have one or some nice features that are worth considering when designing a flexible sensor framework.

3.4.3 Efficiency

Android has very nice feature supporting batch mode. Batch mode helps to increase performance as well as conserve battery and CPU. If batch mode is supported, sensors are allowed to register their events in the batch mode FIFO. Then, developers can call `flush()` function to flush the batched events. It operates asynchronously and a callback function will be called after all events in the batch are delivered successfully. W3C also suggests the same strategy, but is still indecisive.

There is another approach of W3C to increase the efficiency of sensor operation. It is using Stream API of Node.js to deliver a flow of data (accelerometer or electrocardiogram data). Stream API solves general stream functions such as reading from streams and handling issues like reading buffers that is getting full.

In addition, W3C also considered another approach which is not used in any mentioned Sensor APIs. It plans to support cached mode to read cached data when fresh data is not required. This feature helps to preserve sensor's battery life and reduces latency time for accessing cached data instead of reading fresh ones. Personally, I found that it is the easiest and simplest strategy to improve the efficiency of a system.

3.5 Requirements of a good design

In this section, I propose basic requirements of a good sensor framework API design that need to be fulfill to bring benefits down the road.

- The framework should be flexible enough to apply not only to different sensor types but also to different boards, such as: Tessel, Raspberry PI, etc.
- The framework should work with multiple sensors of the same type also.
- The API should provide functions to retrieve necessary information about sensor's capacity.
- The API should fulfill the needs of retrieving all sensor available in the system as well as their statuses.
- It should provide a template for implementing a sensor type.
- Errors should be fully handled.
- It should covers necessary reporting mode for different sensor types.
- It should be designed under users' perspective: easy to learn, easy to use, minimize misunderstanding or misuse, and support enough functions to satisfy requirements.

4. DEFINITION AND IMPLEMENTATION OF THE SENSOR FRAMEWORK API

This chapter describes to the main features of my sensor API framework and its implementation. The sensor framework has been developed in JavaScript and built on top of Node.js [18]. Node.js is the JavaScript runtime environment that uses Google's V8 JavaScript engine. The implementation has been done and tested on version 4.2.1. The framework currently runs on the Raspberry Pi Model B (Rev 2.0, 512Mb) board [19]. That board runs a free operating system named Rasbian. Rasbian is the Debian-based operating system optimized for the Raspberry Pi hardware. The prove-of-concept sensor type I am using for this thesis is the temperature sensor. There are two different temperature sensors I have tested on this framework. The first is the Raspberry Pi's onboard sensor that is used to measure its CPU's temperature. The second is the DS18B20 temperature sensor [20]. Two DS18B20 have been used to test functions related to managing different instances of the same sensor type.

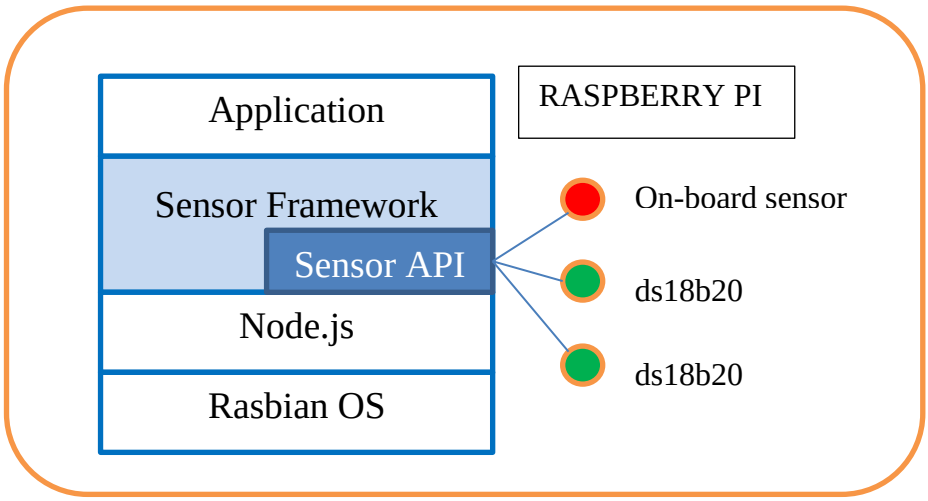

Figure 2. *Sensor framework API - Architecture*

4.1 API Definition

This section presents the API of the sensor framework. This API provides necessary methods to create, manage, and work with different types of sensors.

The whole framework is divided in three main parts:

- `Sensor` class: is a generic sensor object that holds basic information of sensor and provides basic methods to access sensor data as well as retrieve and update sensor attributes.
- `SensorManager` class provides methods to get information about the availability of sensors in the system.
- Sensor category classes: include classes for specific sensor categories such as: `TemperatureSensor`, `AccelerometerSensor`, `GravitySensor` classes, etc. Each class implements necessary data and functions for all sensors belonging to that category.

4.1.1 Sensor class

- `Sensor` class provides general methods that can be applied to all sensors.

Method	Input	Return value	Fired event	Description
start	Object sensor	void	None	Activating the sensor for being ready to use
stop	void	void	None	Deactivating the sensor
resetBuffer	void	void	None	Reset buffer data
readDataBuffer	Number numOfData	Array data	None	Read a specific number of data from the buffer
isValid	Object initOption	String sensorID	'onerror'	Check the validity of the sensor
toString	void	void		Print basic information of the sensor

Table 29. *Sensor framework API - Basic methods of the Sensor class*

- Besides, `Sensor` class also provides a set of getter-setter functions which assists users to update or retrieve attributes' value such as: buffer size, frequency, state, type, etc. Note that, in case users set an illegal value for a sensor's attribute, there is no error event emitted. Instead of that, that illegal value will be ignored, and a notification will be displayed to let user know about that situation.

Method	Input	Return value	Fired event	Description
getType	void	String type	None	Retrieve sensor type
getVendor	void	String vendor	None	Retrieve sensor vendor
getVersion	void	String version	None	Retrieve sensor version
getFreq	void	Number freq	None	Retrieve sensor frequency

setFreq	Number freq	void	None	Set sensor frequency
getThreshold	void	Number threshold	None	Retrieve sensor threshold
setThreshold	Number threshold	void	None	Set sensor threshold
disableThreshold	void	void	None	Disable threshold restriction
getState	void	String state	None	Retrieve sensor state
setState	String state	void	None	Set sensor state
getBufferSize	void	Number size	None	Retrieve buffer size
setBufferSize	Number size	void	None	Set buffer size
getId	void	void	None	Get sensor id

Table 30. *Sensor framework API - Getter/Setter methods in the Sensor class*

- Sensor states: the framework provides three sensor states, including:

States	Description
'idle'	A sensor is in this state when it is either initialized or stopped
'active'	This state indicates that a sensor is working normally
'errored'	A sensor goes to this state whenever an error occurs.

Table 31. *Sensor framework API – Sensor states*

- Event: There is only one event `'onerror'` emitted by a `Sensor` instance. It More events will be presented in the following classes.

Event name	Description
onerror	Emitted when an error occurs

Table 32. *Sensor framework API - Events emitted by the Sensor class*

4.1.2 SensorManager class

- `SensorManager` methods are mainly implemented to support `Sensor` class's demands. However, some functions relating to printing list of sensors or checking whether a specific sensor is supported can be called in the application layer without unexpected effects on the framework.

Method	Input	Return value	Fired event	Description
isSupported	String type, [optional] String id	Boolean	None	Check whether the input sensor type (along with its id if existed) is available in the system, regardless it is using or not.
isUsing	String type, [optional]	Boolean	None	Check whether this sensor type (with specific id) is using
printListOfSensors	void	void	None	Print the list of supported sensors, along with their ids.
printListOfUsingSensors	void	void	None	Print the list of sensors used at that time, along with their ids.
printListOfAvailableSensor	void	void	None	Print the list of available sensors at that time, along with their ids.

Table 33. *Sensor framework API - Public methods in the SensorManager class*

4.1.3 Sensor category classes

First of all, to avoid any ambiguous meaning, some definitions need to be confirmed. In this section, the phrase '*sensor category class'* refers to the class that supports a specific sensor group having the same function. For example, `TemperatureSensor` class provides general functionalities for temperature sensors. Meanwhile, the phrase '*sensor type class*' refers to the class which is exclusively implemented for a specific sensor type (a sensor type always belongs to a sensor category), such as: `TemperatureDS18B20` class supports only for DS18B20 temperature sensors.

- The API of a sensor category class greatly depends on the characteristic of its category. However, all sensor category class provides following methods:

Method	Input	Return value	Fired event	Description
Constructor	Object initOption	void	None	The constructor function of the sensor category class
readData	void	Object data	'onerror', 'ondata', 'onchange'	Read data from the sensor. This function is the core of sensor operation. It is automatically called by the framework at sensor's frequency value after the sensor is initialized. It also can be called by application for one-shot reading.

Table 34. *Sensor framework API - Obliged methods in a sensor category class*

- *`Constructor`*: will be called immediately when application creates an object of this sensor category by calling `new` function. The constructor's input parameter – `initOption` is an object that contains necessary information relating to

the sensor user wants to create. The structure of the `initOption` parameter should be clearly documented.
For example:

```
var tempSensor = new TemperatureSensor({
      type: "temperature_onboard",
      logFilePath: "./logs/onboard.txt"
});
```

- Besides, there are overridden functions declared in `Sensor` class which can be handled in this class if necessary, including:

Method	Input	Return value	Description
start	void	void	start or restart the sensor
stop	void	void	stop the sensor's operation

Table 35. *Sensor framework API – Possible overridden methods in a sensor category class*

- Of course, except those above functions, other functions could be added in this class depending on characteristics and requirements of that sensor category. For example, in the case of temperature sensor, I added the function `setUnit()` that allows user to be able to set their wanted temperature unit, such as Celsius or Fahrenheit unit.
- Events: There are three events emitted by `readData()` function:

Event name	Description
onerror	Emitted when an error occurs
ondata	Emitted when new data is available
onchange	Emitted when new data satisfy threshold restriction

Table 36. *Sensor framework API – Events emitted by a sensor category class*

4.2 API Implementation

This section presents details of how API was implemented. It shows private variables, methods, data structures, and protocols that were used to implement the API. This section also provides more detailed explanation about the sensor discovery strategy used in the framework.

This is an example of files including in the framework:

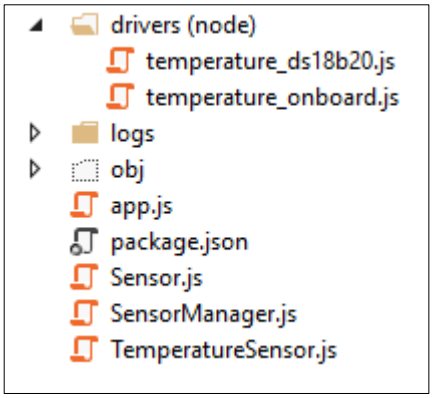

Figure 3. *Sensor framework API – Sample files*

4.2.1 Sensor class

- A private object named `'info'` was defined to contain basic information of associated sensor, including:

ID	Data Type	Default value	Description
type	String	'N/A'	Sensor type
vendor	String	'N/A'	Vendor name
version	String	'N/A'	Sensor version
freq	Number	-1	Sensor frequency (Hz)
threshold	Number	-1	Threshold value(1)
state	String	'idle'	Sensor state

(1) threshold value: if the threshold value is set (not equal to -1), an 'onchange' event will be emitted when the difference of new data and the latest received data is larger than the threshold value.

Table 37. *Sensor framework API - Basic sensor information in 'info' object*

- Besides, `Sensor` class also defines some private variables:

  a. `buffer`: an Array which stores sensor data
  b. `bufferSize`: indicates the size of buffer
  c. `intervalRead`: is an id returned by `setInterval()` function.

  Those variables are only used by inner functions. Users should not modify it.

4.2.2 SensorManager class

- These are private methods that are called by `Sensor`'s methods. These private methods are responsible for checking the validity of a `Sensor` instance based on its type and identification, then register/unregister it in the system to use.

Method	Input	Return value	Description
setListOfSensor	void	void	Initialize list of sensors based on information in the `package.json` file
getIds	String type	Array ids	Retrieve all ids supported by that sensor type
isValid	String type, [optional] String id	Boolean	Check the validity of a sensor type. Errors will be thrown if the sensor type is not supported by the system or it is using. If `isValid()` is passed without any error, this sensor type and id (if existed) is eligible to use.
assignSensor	String type, [optional] String id	Boolean	Register or inform to the system that the sensor is going to be used. The `used` attribute of the sensor in the `listOfSensors` variable will be set to `true` if the sensor is eligible to be used. Appropriated error will be thrown if the sensor is using or unsupported.
unassignSensor	String type, [optional] String id	Boolean	Unregister or inform to the system that the sensor is going to stop using. The re-sources assigned to this sensor will be freed, and `used` attribute of this sensor in the `listOfSensors` variable will be set to `false`. Appropriated error will be thrown if the sensor is not using or unsupported.

Table 38. *Sensor framework API - Private methods in the SensorManager class*

- In the `SensorManager` class, I defined a variable named '*`listOfSensors`*'. This variable holds all information about sensors available in the system. This variable holds an attribute named *`sensors`* which is an array of objects. Each object in this array contains information of each sensor type. Additional attributes could be added if necessary.
- Each object belonging to the `sensors` array contains:

Attributes	Data Type	Description
type	String	Represents the type of sensor
multiple	Boolean	Indicates whether associated sensor type has multiple instances
used	Boolean	Indicates whether the associated sensor is using or not

[optional] list	Array of object	Depend on whether sensor has multiple instances, the `list` attribute is used or not. If there is just only one sensor belonging to this sensor, `list` is discarded. Otherwise, `list` holds an array of objects which has a pair of attributes: `{"id", "used"}`, in which `id` is the sensor id representing the unique identity of this sensor, `used` indicates whether it is using. The former `used` attribute, which goes after `multiple` attribute, will be discards in this case.

Table 39. *Sensor framework API - Object structure in listOfSensors array*

For example: if we have two types of sensors, including: "`temperature_onboard`" and "`temperature_ds18b20`". There is only one instance of "`temperature_onboard`". Meanwhile "`temperature_ds18b20`" has two instances with two distinguished ids. Besides, all instances are not using right now. Then, `listOfSensors` will contain a structure as bellows:

```
{
  "sensors": [
    {
      "type": "temperature_onboard",
      "multiple": false,
      "used": false
    },
    {
      "type": "temperature_ds18b20",
      "multiple": true,
      "list": [
        {
          "id": "28-00000697171e",
          "used": false
        },
        {
          "id": "28-00000697ffc9",
          "used": false
        }
       ]
     }
  ]
}
```

Program 7. *Sensor framework API - listOfSensors sample value*

- **Validity checking protocol**: With the help of information stored in `listOfSensors` property in `SensorManager` class, the task of checking whether a sensor type (with its id if existed) is eligible to use becomes easier. When users want to create a new object for a sensor type, they call `new` function and pass an input parameter containing necessary information relating to the sensor they intend to create. This input parameter will pass to constructor of the sensor type class as `initOption` parameter. Here, the constructor will call `isValid()`

function in the `Sensor` class along with the `initOption`. This parameter will be parsed to retrieve two most important information which are the sensor type and sensor id. Then, these information are passed to the `isValid()` function in `SensorManager` class. The `isValid()` in `SensorManager` class will check that sensor `{type, id}` to see whether it is existed in the system or it is free to use or not. Afterward, it returns its answer back to the upper level.

If the sensor is valid to use, it is assigned in the system and its "`used`" attribute in the `listOfSensors` is updated to `true`. Otherwise, error will be thrown.

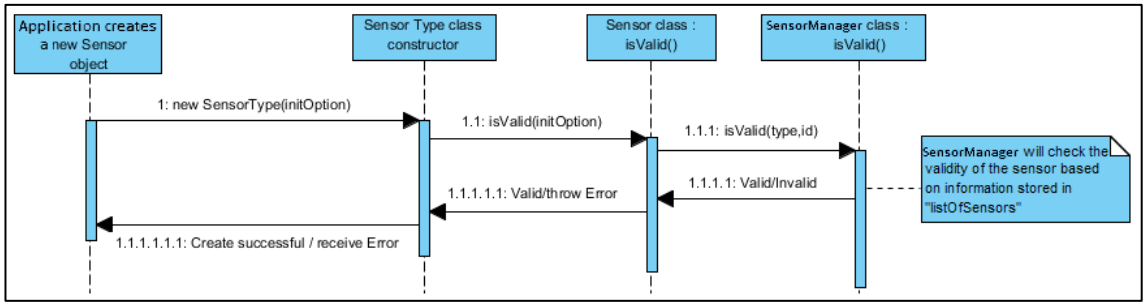

Figure 4. *Sensor framework API - Validity checking sequence diagram*

4.2.3 Sensor category classes

The `Sensor` class plays a role of representing basic information and functionalities of a general sensor. Therefore, it is unable to handle a wide range of sensor categories because they are so different in functionalities, requirements, and data structures. Therefore, each specific sensor category need to be implemented separately, inherited from the pre-constructed `Sensor` class.

Besides, each sensor category may include different sensor types. In that case, drivers of those sensor types need to be provided (placed under the folder named `/driver`). Those files will be specified in the constructor function when application creates this sensor object. Each sensor driver contains all functions that particularly implemented for that sensor type, especially tasks of connecting, disconnecting, and reading data from the lower layer. Then, sensor category class will direct and call appropriate sensor driver according to the applications' need.

Although there are many differences between different sensor category class implementation, they need to implement two mandatory parts: a constructor and `readData()` method.

- Constructor follows some implementation rules as follows:

a. Check whether this sensor (with its id if existed) is valid for using or not. If not, throw an error.
b. If yes, assign appropriate driver for the sensor if there are multi-implementation of the same sensor type existed. For example, there are two kinds of temperature sensor existed in the system. Therefore, the constructor needs to check which kind of sensor user wants to create, and assigns the correct driver for it. An error is thrown if the needed driver is unavailable.

```
if (initOption.type === "temperature_onboard") {
      this.currentTempSensor=require('./drivers/temperature_onboard.js'
).TemperatureOnboard;
} else if (initOption.type === "temperature_ds18b20") {
      this.currentTempSensor=require('./drivers/temperature_ds18b20.js'
).TemperatureDS18B20;
} else {
      throw Error("Driver of " + initOption.type + " is missing");
}
```

Program 8. *Sensor framework API - Sample code for assigning appropriate driver to the sensor*

c. Initialize values for the sensor from data provided by the driver and the user (through `initOption` parameter). Other initialization operations can be implemented here if necessary.

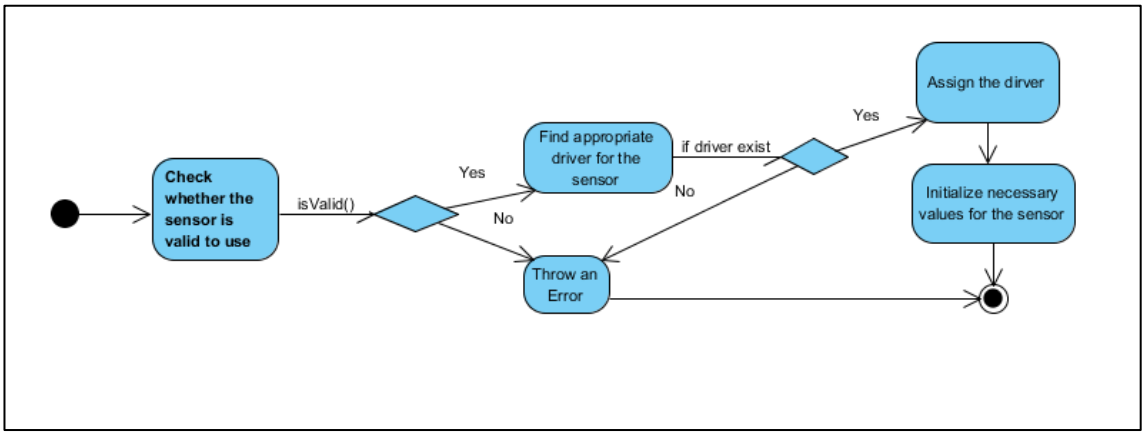

Figure 5. *Sensor framework API - Activity diagram of sensor constructor*

- `readData()`: The mission of this function is to store sensor data in the buffer whenever the data is available. It:

a. Emits `'ondata'` every time new data is available.
b. Checks the threshold value, if the threshold is set, two most recent data should be compared to decide whether that data should be pushed in the buffer. Emitting `'onchange'` event to let application know that there is new data available which satisfies threshold restriction.

c. Emits `'onerror'` event whenever an error occurs, such as: the sensor is unavailable, is stopped without restarting, operates incorrectly, etc.
d. Handle the case when buffer is overflown.

- Lastly, a sensor category class needs to clearly define its sensor data structure. Array in JavaScript can store different data type, even an Object that includes multiple attributes at the same time. Therefore, `buffer` array using to store sensor data is very flexible. In my implementation, data received from the temperature sensors is an object having two attributes: {`value`, `timeStamp`}. It is important to clarity beforehand the structure of sensor data stored in the buffer. Hence, users can parse it correctly.
- Because sensor identification number (id) is retrieved from the lower layer, such as middleware or physical layer. Therefore, in case there are more than one instances of the same sensor type existed, driver needs to implement a function to get ids of instances. Then, sensor category class can call that function to retrieve the sensor id. This function is also compulsory for the sensor discovery task (which will be mentioned in the next section 4.2.4).

4.2.4 Sensor discovery

Although sensor discovery is out of the scope of my thesis, but I found it's worth mentioning. It's important to raise the question of how does the framework learn and gather information about all sensors available in the system when it starts and while it is running. There is no simple answer for this question, and it seems too complicated to solve it in the hardware layer. Therefore, I proposed a quite simple solution for this issue instead, even although I do not claim that it is an optimum strategy.

I use a JSON file name `package.json` to store all sensor types available in the system. When a new sensor type is added in the system, programmer must add that sensor type in this file. Otherwise, the system cannot recognize it when it runs.

In the beginning stage, when the system runs, `Sensors` object is constructed automatically, along with its property `listOfSensors`. To initialize `listOfSensors`, the `package.json` file is parsed and the list of all sensor types is stored in that variable. `listOfSensors` is the heart of sensor discovery strategy which is used to manage all sensors in the system. However, `package.json` only holds the list of sensor type. It lacks of other important information, which will be added later.

So now, after parsing the `package.json`, each sensor type will be added two more attributes. First is the property `'multiple`, which indicates whether this sensor type has multiple instances. Second is the property `'used'`, which says whether this sensor type is used or not. Both two attributes are initialized with `false` value. Then, the pro-

gram will go through each sensor type and check if it has more than one instance. If yes, `multiple` will be set to `true`, the `used` variable is deleted. Ids of instances will be added in an array. Each instance has a `used` attributed initialized to `false`. After all, we have the list of all sensor type along with necessary information that help us distinguish whether it is a single or a multiple sensor type, and its operation state as well. (Program 7 describes an example of data contained in `listOfSensor` property.)

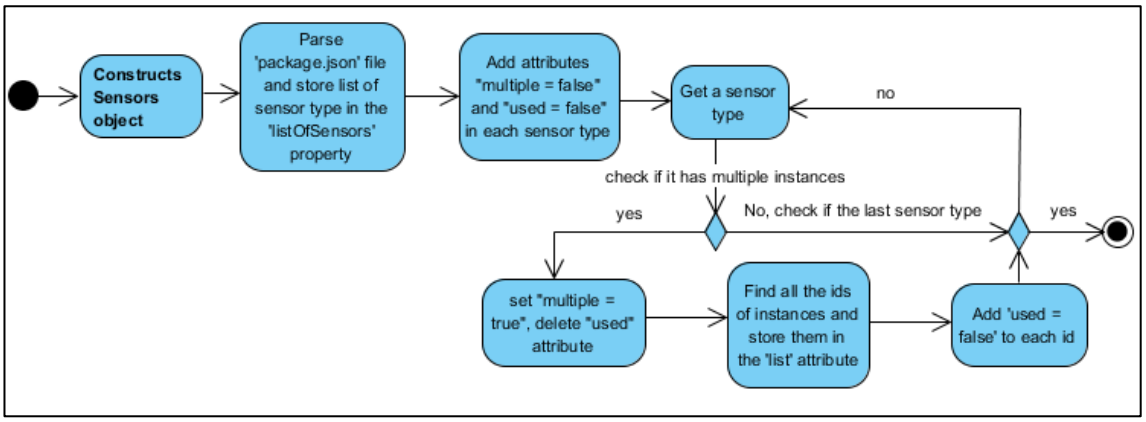

Figure 6. *Sensor framework API - Sensor discovery diagram*

In my program, Raspberry Pi's on-board temperature has only instance as default. However, DS18B20 has more than one instances working concurrently. Each DS18B20 sensor has its own id. When a DS18B20 sensor connects to the main board, a new folder is created by the operating system under the path: */sys/bus/w1/devices*, with exclusive named: 28-xxxxxxx with last 7 digits is the id of that sensor [24]. For example, I connected two DS18B20 sensors into my Raspberry Pi board, then, I got two folders as you can see in the picture below.

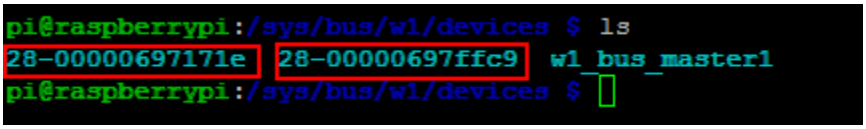

Figure 7. *Sensor framework API - Folders under /sys/bus/w1/devices*

Therefore, I retrieved ids of different DS18B20 by following these steps:

Traverse through each folder under the path: */sys/bus/w1/devices*.

a. Check if the folder name starting with "28-" which is unique for 1-wire address.
b. If yes, push the id in the array. If no, continue to the next folder.

(If there is no folder name starting with "28-", a returned empty array will let program know that there is no DS18B20 connected in the system.)

Depending on each sensor type, we develop different strategies to figure out whether it is the sensor type with unique instance or multiple one, and how to retrieve id of each sensor instance if needed.

5. INSTRUCTIONS TO USE THE SENSOR FRAMEWORK API

There are two important stakeholders of this project. The first is application writers who directly use the API in their own code to implement their application. The second is sensor providers who add their sensors in the framework. Hence, applications can utilize them. This chapter provides details of instructions for those stakeholders. It is divided into two parts. The first part is instructions of writing an application using the sensor framework API. The second part guides sensor provides in extending the sensor framework to work with their sensors.

5.1 How to write an application?

In the beginning of each application, appropriate sensor modules need to be loaded for using. For example:

```
var TemperatureSensor = require('./TemperatureSensor.js').TemperatureSensor;
var ProximitySensor = require('./ProximitySensor.js').ProximitySensor;
```

Besides, the `SensorManager` module can be loaded if the user intends to use functions supported by `SensorManager` class.

```
var SensorManager = require('./SensorManager.js').SensorManager;
```

Create a sensor object

Application creates a sensor object by using function `new`. Input of the constructor is an object holding required information is needed to construct this sensor. Some properties that should be declared are:

- `type`: to identify the sensor type. It is a mandatory property.
- `'id'`: to identify the sensor. It is an optional property. In case users have no idea about the id of sensor they want to use, this information can be ignored. The framework will automatically choose a free id (if exist) and assign it to that sensor. If there is no id available at that time, an error will be thrown.

The `type` in input parameter object of the `new` function must be exactly identical to the `type` declared in `package.json`. Otherwise, the system cannot recognize and see it as undefined sensor. Fortunately, `SensorManager` provides `printListOfAvailableSensors` function which can help developers check which sensors

are available in the system. Type and id (if existed) of available sensors are printed (Figure 8). Therefore, developers can use that information to create their needed sensor object. If successful, the constructor will return the sensor object and assign it to the declared variable. Afterward, developer can use that variable to access to the sensor.

Figure 8. *Sensor framework API – List of available sensors display sample*

Apart from the compulsory information as above, application developers can add any other information describing their requirements. It is important for sensor providers to clearly declare and document all necessary requirements about input parameters passed to the constructor, both names as well as their purposes. For example, in the example I developed to test my framework, the structure of input parameter object of `TemperatureSensor` contains information as follows:

- `type`: sensor type, must be: "`temperature_onboard`" or "`temperature_ds18b20`"
- `id`: (optional) sensor id.
- `logFilePath`: (optional) path of the log file. If it is declared, sensor data will be logged automatically in a file indicated by this attribute.

Events

- Because Sensor class is extended from `EvenEmitter` class of Node.js, therefore, each sensor object can emit appropriate events by itself. There are three events emitted by every sensor. Developers can use it to access sensor data and handle errors:

Event name	Description
ondata	Emitted when new sensor data is available
onchange	Emitted when new sensor data satisfies threshold requirement.
onerror	Emitted when an error occurs

Table 40. *Sensor framework API - Events*

- Application captures and handles events by using the asynchronous function call `'on'`. For example:

```
sensor.on('ondata', function (event) { //event handler });
```

Note: The sensor is automatically started after creating, thus user do not need to call `start()` function to start it. However, it must be called if user wants to restart using it after stopping it by calling `stop()` function.

If there are multiple instances of the same sensor type, the only way to distinguish or select them is using their ids. Developers can query list of available sensors (type + ids) by calling: `printListOfAvailableSensors()` from the `SensorManager` class. If developers do not specific sensor's id in their input parameter, the framework will automatically assigned any free sensor of that type in the system (if existed) to the variable. Unfortunately, the framework does not provide other strategies that help developers to select sensors based on other means such as location.

Example

```
var TemperatureSensor = require('./TemperatureSensor.js').TemperatureSensor;
var SensorManager = require('./SensorManager.js').SensorManager;

// Temperature onboard object
var tempOnboard = new TemperatureSensor({
    type: "temperature_onboard",
    logFilePath: "./logs/onboard.txt"
});

// The 1st DS18B20 temperature sensor object
var tempDS18B20 = new TemperatureSensor({
    type: "temperature_ds18b20",
    logFilePath: "./logs/ds18b20.txt"
});

// The 2nd Temperature onboard object
var tempDS18B20_2 = new TemperatureSensor({
    type: "temperature_ds18b20",
    logFilePath: "./logs/ds18b20_2.txt"
});

// Print list of supported sensors
SensorManager.printListOfSensors();
SensorManager.printListOfUsingSensors();
SensorManager.printListOfAvailableSensors();

// Print basic information about sensors
tempDS18B20.toString();

console.log("Temperature data will be log automatically in files under ./logs
folder");

/*----------- Event handlers -----------*/
tempOnboard.on('ondata', function (event) {
    console.log('onBoard: [' + event.timeStamp + "]: " + event.value);
});

tempOnboard.on('onerror', function (err) {
    console.log(err);
```

Here, `tempDS18B20` and `tempDS18B20_2` are automatically assigned different ids by the framework. Application can retrieve them by calling `getId()` function.

```
});

tempDS18B20.on('onchange', function (event) {
    console.log('ds18b20: [' + event.timeStamp + "]: " + event.value);
});

tempDS18B20.on('onerror', function (err) {
    console.log("tempDS18B20 has error");
    console.log(err);
});

tempDS18B20_2.on('onchange', function (event) {
    console.log('ds18b20: [' + event.timeStamp + "]: " + event.value);
});

tempDS18B20_2.on('onerror', function (err) {
    console.log("tempDS18B20 has error");
    console.log(err);
```

Program 9. *Sensor framework API - Application sample code*

```
pi@raspberrypi:~/MyCode $ node app.js
List of supported sensors:
----------------------------
temperature_onboard
temperature_ds18b20
id[0] = 28-00000697171e
id[1] = 28-00000697ffc9
----------------------------
List of sensors are using:
----------------------------
temperature_onboard
temperature_ds18b20
id[0] = 28-00000697171e
id[1] = 28-00000697ffc9
----------------------------
List of sensors are available:
----------------------------
None
----------------------------
temperature_ds18b20
        {Id: 28-00000697171e
        Vendor: Maxim Integrated
        Version: N/A
        Wakeup: false
        Frequency: 0.5
        Threshold: -1
        Timeout: -1
        Unit: C}
Temperature data will be log automatically in files under ./logs folder
onBoard: [Thu Jan 12 2017 13:14:49 GMT+0000 (UTC)]: 43.2
ds18b20: [Thu Jan 12 2017 13:14:49 GMT+0000 (UTC)]: 25.13
ds18b20_2: [Thu Jan 12 2017 13:14:50 GMT+0000 (UTC)]: 24.81
onBoard: [Thu Jan 12 2017 13:14:52 GMT+0000 (UTC)]: 43.2
ds18b20: [Thu Jan 12 2017 13:14:53 GMT+0000 (UTC)]: 25.13
ds18b20_2: [Thu Jan 12 2017 13:14:54 GMT+0000 (UTC)]: 24.81
onBoard: [Thu Jan 12 2017 13:14:55 GMT+0000 (UTC)]: 43.2
ds18b20: [Thu Jan 12 2017 13:14:56 GMT+0000 (UTC)]: 25.13
ds18b20_2: [Thu Jan 12 2017 13:14:58 GMT+0000 (UTC)]: 24.81
```

Figure 9. *Sensor framework API – Console output of the sample code*

```
1 [Thu Jan 12 2017 13:14:49 GMT+0000 (UTC)]: 43.2
2 [Thu Jan 12 2017 13:14:52 GMT+0000 (UTC)]: 43.2
3 [Thu Jan 12 2017 13:14:55 GMT+0000 (UTC)]: 43.2
4 [Thu Jan 12 2017 13:14:58 GMT+0000 (UTC)]: 42.7
5 [Thu Jan 12 2017 13:15:02 GMT+0000 (UTC)]: 43.2
```

Figure 10. *Sensor framework API – Log file onboard.txt*

5.2 How to add a new sensor in the framework?

The purpose of building this framework is to create a convenient environment for application to work with different types of sensor, in a more consistent way. In this case, under the users' perspective, users only need to change the input parameter in the constructor function when they want to create any sensor object and start working with it.

For example:

```
var temperature = new TemperatureSensor ({type: "temperature_onboard"});
var proximity = new ProximitySensor ({direction: "rear", side: "left"});
var geolocation = new GeolocationSensor ({accuracy: "high"});
```

In addition, the framework also provides convenient strategies to retrieve data from the sensor: through the events (`onchange` or `ondata`), or buffer, or direct reading. All sensor type registered in the system can use their appropriate methods to get the data which are provided by the framework.

To achieve these advantages, the process of extending new sensor type in the framework needs to obey these following instructions:

- Add the new sensor type name in the `package.json` file.

Sensor category file

- Create a file for new sensor category, for example `CategorySensor.js`.
- In this file, create a `CategorySensor` class which inherits from `Sensor` class:
    a. Load module `Sensor` by using `'require'` function:

    ```
    var Sensor = require('./Sensor.js').Sensor;
    ```

    b. Extend current class from the `Sensor` class:

    ```
    CategorySensor.prototype = Object.create(Sensor.prototype);
    CategorySensor.prototype.constructor = CategorySensor;
    Object.assign(CategorySensor.prototype, Sen-
    sor.super_.prototype);
    ```

- Implement the constructor with input parameter is `initOption`. (You can use another name but I recommend using the same name for all sensor category classes to make the framework more consistent). In the constructor, do these tasks:
    a. Apply `Sensor` to `this` for inheritance:

    ```
    var CategorySensor = function (initOption) {
        Sensor.apply(this, arguments);
        ...
    }
    ```

    If `Sensor` doesn't apply to `this`, modification of any properties of one instance will also be applied to those properties of others instances (in case more than one instances of `CategorySensor` are created).
    b. Check whether the sensor is valid to use by calling:

    ```
    var sensorId = this.isValid(initOption);
    ```

    The `isValid()` function will return the `sensorId` if available. We add this value in the `initOption` to keep it for later use:

    ```
    if (sensorId) {
        initOption.id = sensorId;
    }
    ```

    c. Check the state of sensor, return if the state is `errored`

    ```
    if (this.getState() === 'errored') return;
    ```

    d. If this sensor type has many drivers, assign the appropriate driver for it based on the `type` value in the `initOption`. (See the sample code program 8).
    Throw an error if needed driver is missing.
    e. Add other initialization code if necessary.
    f. At the end of the constructor, call the function `start()` to start the sensor:

    ```
    this.start(this);
    ```

- Implement the function `readData()`. Although implementation of reading sensor data depends on each sensor type's characteristic. However, all the `readData()` need to obey some rules as follows:
    a. At the beginning, check the current state of the sensor. If the state is `'errored'` or `'idle'`, emit `'onerror'` event.
    b. Check threshold restriction if needed. Emit event `'onchange'` for data satisfy threshold restriction.

c. Emit `'ondata'` event whenever a new data is available.
d. Emit `'onerror'` event whenever an error occurs.
e. Handle buffer overflown.

- Implement other necessary functions relating to that kind of sensor if necessary.
- Include any driver if necessary.
- Export the `CategorySensor` module:

```
module.exports.CategorySensor = CategorySensor;
```

Sensor driver files

Note: If you are sure that there will be only one sensor belonging to this sensor category, no driver needs to be implemented. All functions relating to that sensor type can be implemented only in the sensor category file. Sensor driver files are required only in case you have different implementation for different sensor types that belongs to a sensor category.

- Any sensor driver file should be placed under the folder *./drivers*.
- Store any needed information in the `info` variable. This information will be added in the sensor along with the information in `initOption` object when the sensor is instantiated. The `info` is a part of the driver, it should add basic information relating to the sensor such as: type, id, vendor, frequency, unit, etc. If there is only sensor type belonging to a sensor category, the `info` object should be declared in the sensor category file because no driver file is needed in that case.
  For example:

```
/** @var {Object} info - Hold DS18B20's information */
TemperatureDS18B20.info = {
    type: "temperature_ds18b20",
    id: "",
    vendor: "Maxim Integrated",
    freq: 0.5, //Hz
    unit: 'C', //Celsius Degree as default
};
```

- If there is multiple sensors of the same type, implement function named `getIds()` to retrieve all identification numbers of all instances available in the system. An array holding ids must be returned. Otherwise, if it only has one instance, this function can be ignored. Because getting ids of sensors depends on the implementation of each sensor type, there is no default implementation for this task.
- Implement `readData()` to read sensor data because each sensor type retrieves its data in its own way.
- Export the module.

Update `getIds()` in `SensorManager` class

- Update `getIds()` function for new sensor type in `SensorManager` class:

```
SensorManager.getIds = function (type) {
    if (type === "temperature_ds18b20") {
        return TemperatureDS18B20.getIds();
    }
    else if (type === “temperature_newtype”) {
        // Add your code here
    }
    return null;
}
```

Program 10. *Sensor framework API – Sample code for updating getIds() function in the SensorManager module*

Naming convention

Besides, there are some conventional rules relating to naming. These rules create an overall standard of the framework, including:

- Sensor category class: should be named in upper camel case with the suffix *Sensor*. The first part represents the category of the sensor.

  Ex: `TemperatureSensor`, `ProximitySensor`, etc.
- Sensor type file: should be named in lowercase with the first part is the sensor category; second part is the specific type of that sensor. Two of them are separated by an underscore character.

  Ex: temperature_onboard.js, temperature_ds18b20.js, etc.
- Sensor driver class: should be named in upper camel case with the first part is the sensor category; second part is the specific type of that category. The type part should be written in uppercase. There should be compatibility between the driver file name and the driver class name.

  Ex: `TemperatureOnboard`, `TemperatureDS18B20`, etc.

6. RESULTS

In this part of the thesis, I present the evaluation of my design and implementation. The evaluation is divided into two different parts. The first part is devoted to assessing my work, considering pros and cons of my sensor framework API. The second part is to present the API's testing phrase which has been done so far.

6.1 Assessment

As mentioned before, there are some basic requirements of a good sensor design proposed in section 3.4. The assessment are based on those requirements which helps to reveal major advantages and disadvantages of the project.

Advantages

- Simple: the design is really simple with only few main classes. Each class is in charge of separated parts. Therefore, it is easy to understand, manage, and maintain.
- Easy to use: because the aim of the sensor framework is to provide a convenient environment for users to access different sensors in the system. Hence, working with the framework is undemanding and easy. Users can use the framework to achieve their purposes without great effort and difficulties, with fewer code line than usual. Besides, the framework provides enough functions satisfying users' demands, adequate error handling, and notifications, which contributes to mitigate the difficulty of application developing on the users' side.
- Easy to add new sensor type: the framework provides a straightforward template to add a new sensor type. Clear instructions are given, indicating clearly which parts are obliged to be implemented, which parts are optional, which classes should be inherited, etc. Hence, extending new sensor types in the framework is quite simple and easy.
- Flexible: the framework works with not only different sensor type but also different instances of the same sensor type. It allows coexistence of different implementation or drivers of sensors belonging in the same type. All of them bestows reasonable flexibility on the framework.
- It allows sensors removed/re-connected at the runtime without affecting on the operation of other sensors. The interruption handler is implemented in the sensor category or sensor driver layer. Developers can choose either delivering non-

measured data (such as zero) or totally stopping the sensor operation while it is removed from the system.

Disadvantages

- Currently, the sensor framework does not support different boards. Appropriate changes need to be done when migrating to other baseboards.
- The framework provides only one strategy to select/distinguish multiple sensor instances of the same type, which is using their ids. It is much more convenient if developers are able to select different sensor instances based on their locations, directions, or other ways.
- The implementation of upper layers such as `Sensor` and `SensorManager` still has some parts depending on the implementation of specific sensor categoary classes. For example, developers need to update `getIds()` function in `SensorManager` class whenever they want to add new sensor category in the framework. This dependency decreases the flexibility, maintainability, and scalability of multilayered architecture.
- The API does not take advantage of method chaining. This is a useful technique that allows calling multiple functions consecutively. This technique will help to simplify the code.
- Some manual works should be automated, such as: updating `getIds` function for new sensor type in `SensorManager` class.
- The framework depends heavily on the `'package.json'` file. If there is any problem with that file, the whole system is broken down.
- Security is not taken into account in designing and implementation of the framework. The framework does not support any encrypted data transaction. Therefore, the framework is vulnerable and is easy to be attacked by malicious activities. Information transferred is insecure and unprotected.

  However, dealing with sensor discovery, bootstrapping, security and troubleshoot problem is out of the scope of this thesis. Therefore, these weaknesses would be passed to the next phrase rather than be considered here.

6.2 Testing

The code was written in JavaScript and tested on:

- Board: Raspberry Pi Model B, Rev 2.0, 512Mb.
- Sensors:
  a. Onboard temperature sensor
  b. 2 DS18B20 temperature sensors
- Platform: Node.js version 4.2.1

- Test method: Unit test

These are some statistics on test results:

Class	Number of methods	Number of tested methods
Sensor	18	17/18 (`readData()` is an abstract function)
SensorManager	10	10/10
TemperatureSensor	4	4/4
TemperatureDS18B20	2	2/2
TemperatureOnboard	2	2/2

Table 41. *Testing - Statistic on number of tested methods*

	Use cases	Result
1	Work with only onboard temperature sensor	Pass
2	Work with only one DS18B20 sensor	Pass
3	Work with 2 DS18B20 sensors	Pass
4	Work with onboard temperature sensor and one DS18B20	Pass
5	Work with onboard temperature sensor and one DS18B20	Pass
6	DS18B20 with correct specific id	Pass
7	DS18B20 with incorrect specific id	Pass
8	DS18B20 without id specification	Pass
9	DS18B20s removed and re-connected suddenly	Pass
10	Work with different frequency	Pass
11	Display in Celsius and Fahrenheit units	Pass
12	Define illegal temperature unit	Pass
13	Unknown sensor	Pass
14	Declare more than 2 instances of DS18B20	Pass
15	Declare more than one instances of onboard temperature	Pass
16	Not using log file path	Pass
17	Use illegal log file path	Pass
18	Stop a sensor and try to retrieve data without restarting it	Pass
19	Use threshold value	Pass
20	Driver of sensor is missing	Pass

Table 42. *Testing - Main test cases which are already passed*

	Use cases
1	Test on other Raspberry Pi board versions
2	Test with more than 2 different temperature sensor types
3	Test on other Node.js version
4	Buffer overflow case

Table 43. *Testing – Test cases which are untested*

Currently, the framework is quite small and simple. Then, it is possible to test all those test cases manually. However, the API can be greatly expanded in the future with more functions and classes included. It will create a growing number of tests. Consequently, an automatic testing strategy should be applied such as UnitJS, Jasmin, etc. Those testing tools help to build a more robust and reliable framework in the future.

7. CONCLUSION

In conclusion, the development of Internet of Things brings considerable opportunities as well as possible challenges to developers and researchers. Regarding to sensor-related aspect, IoT systems are required to sense increasing complex environments. Thus, IT needs to offer a variety sensing technologies to support the diversity of IoT applications. Therefore, a good foundation for sensor operation is inevitable.

A comprehensive and consistent sensor framework brings huge benefits to the IoT development. This issue begins to draw attention from communities and research organizations. There are different approaches in building a sensor framework API. Each of them has their own pros and cons. Digging deeply in existing sensor libraries brings necessary knowledge and understanding about major requirements of a good design as well as appropriate techniques to implement them.

In fact, design a good sensor framework API that covers a wide range of sensor types is not an easy task. Compromises between objectives such as efficiency and easy to implement, flexibility and simplicity, etc. need to be settled when designing. There is no perfect solution.

This thesis proposed a design and implementation of a sensor framework API. This API draws a solution to cope with the inconsistence and incompatibility of a variety of sensor APIs. This sensor framework API borrowed some nice features from different other sensor libraries that had been researched. Although this design implies some drawbacks, it has many good features as well. A complete framework is much more complex and it requires many researches and effort. Nevertheless, it is surely worthy of endeavor.

7.1 Future work

Currently, the framework runs only on Raspberry Pi, which makes it heavily depend on that hardware. The next target might be improving the framework to make it become independent on a specific hardware. The framework should be able to run on different physical boards, especially single-board computers such as Tessel, Phidget, Arduino, or Edison, etc.

In addition, current framework only works with temperature sensors as a proof-of-concept. However, a framework supporting wide range of sensor categories is what makes it usable. Besides, the process of extending variety of sensor categories will contribute to improve the design of the framework itself.

Last but not least, a complete framework needs to consider different other issues such as bootstrapping, troubleshooting, or security, etc. to make the framework more robust, reliable, and secure. The current solution of sensor discovery using in this framework is heavily dependent on `package.json` file, which makes the framework vulnerable. Mitigating this dependency needs to be considered to get rid of this drawback. These problems can be addressed in the future to improve the framework.

REFERENCES

[1] Internet Society Organization, “The Internet of Things: An Overview”, pp. 10, October 2015.

[2] K. Ashton, “That ’internet of things’ thing in the real world, things matter more than ideas”, RFID Journal, June 2009.

[3] C. Elena-Lenz, Internet of Things: Six key characteristics, July 2015 [Online]. Available: https://www.linkedin.com/pulse/internet-things-six-key-characteristics-carlos-elena-lenz. [Accessed on: 01-08-2016]

[4] Libelium, “50 sensors Applications for a Smarter World”, [Online]. Available: http://www.libelium.com/resources/top_50_iot_sensor_applications_ranking/ [Accessed on: 02-08-2016]

[5] S. Sam, “Internet of Things connected devices to almost triple to over 38 billion units by 2020”, Juniper Research, July 2015 [Online]. Available: http://www.juniperresearch.com/press/press-releases/iot-connected-devices-to-triple-to-38-bn-by-2020 [Accessed on: 07-03-2016]

[6] “Global Markets and Technologies for Sensors”, BBC Research, pp 61-72, July 2014.

[7] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, D. Aharon, “Unlocking the potential of the Internet of Things”, Mc Kinsey Global Institute, pp.62, June 2015.

[8] Wikipedia, “Sensor”, [Online]. Available: https://en.wikipedia.org/wiki/Sensor [Accessed on: 05-08-2016]

[9] W3C, “Generic Sensor API”, [Online]. Available: https://w3c.github.io/sensors/ [Accessed on: 10-08-2016]

[10] B. N. Miller, “Phigets API”, GitHub [Online]. Available: https://github.com/RIAEvangelist/node-phidget-API [Accessed on: 15-08-2016]

[11] Tessel, “Tessel API”, GitHub [Online]. Available: https://tessel.io/opensource [Accessed on: 15-08-2016]

[12] Insteon Hub, “Insteon Hub API”, GitHub, [Online]. Available: https://blog.automategreen.com/post/insteon-hub-sensor-api/ [Accessed on 15-08-2016]

[13] https://github.com/automategreen/home-controller [Accessed on 15-08-2016]

[14] Android, “Sensor Overview”, [Online]. Available: http://developer.android.com/guide/topics/sensors/sensors_overview.html [Accessed on 15-08-2016]

[15] Apple Inc., “Core Motion”, [Online]. Available: https://developer.apple.com/reference/coremotion [Accessed on 16-08-2016]

[16] W3C, “Sensor issues”, GitHub, [Online]. Available: https://github.com/w3c/sensors/issues [Accessed on 30-08-2016]

[17] F. Ahmadighohandizi, K. Systä, “Application Development and Deployment for IoT Devices”, Department of Pervasive Computing, Tampere University of Technology, pp.3-5, 2016.

[18] Node.js, [Online]. Available: https://nodejs.org/en/ [Accessed on: 13-09-2016]

[19] Raspberry Pi, “Raspberry Pi 1 Model B”, [Online]. Available: https://www.raspberrypi.org/products/model-b/ [Accessed on: 13-09-2016]

[20] MaximIntegrated, “DS18B20 Programmable Resolution 1-Wire Digital Thermometer”, [Online]. Available: https://www.maximintegrated.com/en/products/analog/sensors-and-sensor-interface/DS18B20.html [Access on: 13-09-2016]

[21] Adafruit, “Sensor Calibration”, [Online]. Available: https://learn.adafruit.com/calibrating-sensors/so-how-do-we-calibrate [Access on: 05-08-2016]

[22] Electrical4U, “Types of Sensors”, [Online]. Available: http://www.electrical4u.com/sensor-types-of-sensor/ [Access on: 05-08-2016]

[23] W3C, [Online]. Available: https://www.w3.org/ [Access on: 10-08-2016]

[24] S. Monk, “Adafruit’s Raspberry Pi lesson 11. DS18B20 Temperature Sensor”, Adafruit Industries, pp. 11, March 2016.